Inter-process Concolic Testing of Full-stack JavaScript Web Applications

Maarten Vandercammen

Dissertation submitted in fulfilment of the
requirement for the degree of Doctor of Sciences

October 13, 2023

Promotor:
Prof. Dr. Coen De Roover, Vrije Universiteit Brussel

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel : +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN : 9789464443806
NUR CODE : 980
THEMA : UMZT

Abstract

Web applications are becoming increasingly prevalent. Example applications include collaborative text editors and drawing applications. We define *full-stack JavaScript web applications* as web applications of which both the client and the server have been implemented in JavaScript.

Several automated testing approaches have been proposed to verify the correctness of sequential, non-distributed applications. Prominent among these is *concolic testing*, which systematically explores all execution paths through the program by collecting symbolic constraints over the program inputs.

We transpose concolic testing to the domain of full-stack JavaScript web applications, which gives rise to several challenges unique to these systems. For example, both client and server processes are event-driven, so concolic testers for these applications must craft elaborate event sequences to explore some parts of the program. Furthermore, because of the interconnected nature of these processes, the execution of one process may affect that of another in unexpected ways.

We propose an approach to concolic testing for these types of applications that addresses these challenges. This approach relies on performing *inter-process* testing of the application, which tests all instances of the client and server processes while observing their communication. Inter-process testing hence preserves information flow between processes, thereby increasing precision and preventing false positive errors. Inter-process testers stand in contrast to *intra-process* testers, of which the execution paths do not cross the boundary of a process. We implement inter-process testing in a novel concolic tester called *StackFul*.

StackFul also considers the event-driven nature of full-stack JavaScript web applications. Event-driven code gives rise to the problem of *state explosion*, where an exponential number of states are created while testing the application. To solve this problem, we introduce a novel form of *state merging* for concolic testing of event-driven applications that reduces the number of states by merging together similar states.

We evaluate StackFul on eight real-world applications. We measure to what extent StackFul is capable of i) covering execution paths, ii) finding errors on the server of these applications, and iii) discerning high-priority from low-priority server errors. Furthermore, we evaluate the impact of incorporating state merging. We show that state merging almost always requires fewer test runs to achieve higher code coverage.

Samenvatting

Webapplicaties worden steeds populairder. Voorbeelden van dit soort applicaties zijn onder andere collaboratieve tekstverwerkingsprogramma's en tekenapplicaties. Wij definiëren *full-stack JavaScript webapplicaties* als webapplicaties waarvan zowel de client als de server geïmplementeerd werden in JavaScript.

Er bestaan verschillende geautomatiseerde testtechnieken om de correctheid van sequentiële, niet-gedistribueerde applicaties te verifiëren. Prominent hierin is *concolic testing*, welke systematisch alle executiepaden doorheen een programma test door symbolische constraints te verzamelen over diens programma-inputs.

We verplaatsen concolic testing naar het domein van full-stack JavaScript webapplicaties. Hierbij duiken verschillende uitdagingen op die uniek zijn aan dit soort systemen. Aangezien bijvoorbeeld zowel het client- als het server-proces event-driven zijn, moeten concolic testers van deze applicaties ingewikkelde sequenties van events aanmaken om bepaalde delen van het programma te testen. Bovendien kan, omwille van de sterk geconnecteerde aard van deze processen, de uitvoering van het ene proces de uitvoering van een ander op onverwachte manieren beïnvloeden.

We stellen een aanpak van concolic testing van deze applicaties voor die zich richt op deze uitdagingen. Deze aanpak maakt gebruik van *inter-proces* testing, waarbij alle instanties van client- en server-processen getest worden en hun onderlinge communicatie tegelijkertijd geobserveerd wordt. Inter-proces testing behoudt de informatiestroming tussen processen, waardoor precisie wordt verbeterd en vals-positieve fouten worden vermeden. Inter-proces testers onderscheiden zich van *intra-proces* testers, wiens executiepaden niet de grenzen van een proces overschrijden. We implementeren inter-proces testing in een nieuwe concolic tester genaamd *StackFul*.

StackFul houdt ook rekening met de event-driven aard van full-stack JavaScript webapplicaties. Event-driven code geeft aanleiding tot het probleem van *state explosion*, waarbij een exponentieel aantal staten wordt aangemaakt tijdens het testen van de applicatie. Om dit probleem op te lossen, introduceren we een nieuwe vorm van *state merging* voor concolic testing van event-driven applicaties, waarbij het aantal staten beperkt wordt door gelijkaardige staten samen te voegen.

We evalueren StackFul op acht full-stack JavaScript webapplicaties. We meten in hoeverre StackFul in staat is om i) hun executiepaden te verkennen, ii) fouten te vinden in de server van deze applicaties, en iii) hoog-prioritaire fouten te onderscheiden van laag-prioritaire serverfouten. Daarnaast evalueren we de impact van state merging. We tonen aan dat state merging bijna altijd minder testiteraties vereist om een groter deel van de applicatie te verkennen.

Acknowledgements

I would like to start these acknowledgements by expressing my gratitude to my promotor, Coen De Roover, who supported me throughout my PhD and without whom I would not have been able to complete it. Coen convinced me to start a PhD and gave me the opportunity to keep working on my research, while devoting a large part of his available time on helping me convert vague ideas into concrete realisations. Coen, thank you for your help and encouragement over these past years!

I would also like to thank the members of my jury, Beat Signer, Cristian Cadar, Kris Steenhaut, Joeri De Koster, Viviane Jonckers, and Xavier Devroey, for taking the time to read this dissertation and for providing insightful comments and feedback that have helped improve the quality of this text.

Although everyone at Soft deserves an elaborate thank-you in these acknowledgements, I will have to keep it short. Some of the many people who deserve a special shout-out, however, include: Tim, for your saintly patience and support, both when we were still sharing an office and afterwards; Mathijs, for helping to keep me (mostly) sane through the final stretches of my PhD; Christophe, simply for being Christophe; Laurent, without whom there would be no StackFul; Jens, for your welcome distractions and for listening to my rants about state merging; Sam, for your strong opinions on food and other things; Quentin, for your thoughtful feedback on my work; Cindy and Johannes, for your help with supervising students and managing projects while I was distracted writing this dissertation; and the whole rest of Soft for turning 10F into a unique and enjoyable work environment.

Ten slotte wil ik ook mijn ouders, grootouders en peetouders, Pieter, Leen en de kinderen, en de hele rest van de familie bedanken om mij doorheen alle jaren te steunen. Zonder jullie had ik hier niet kunnen staan!

Contents

List of Acronyms

API Application Programming Interface.
BSE Backwards Symbolic Execution.
CES Core Event Semantics.
CESK Control-Environment-Store-Continuation.
CSS Cascading Style Sheets.
DAG Directed Acyclic Graph.
DOM Document Object Model.
GUI Graphical User Interface.
HF `HandlerFinished`.
HTML HyperText Markup Language.
HTTP Hypertext Transfer Protocol.
IO Input/Output.
ITE If-Then-Else.
JIT Just-In-Time.
JSON JavaScript Object Notation.
LAMP Linux, Apache, MySQL, PHP.
LOC Lines Of Code.
MEAN MongoDB, Express, Angular, Node.js.
MPI Message Passing Interface.
PC Path Constraint.
SDK Software Development Kit.
SMT Satisfiability Modulo Theories.
SPMD Single Program, Multiple Data.
SQL Structured Query Language.
UI User Interface.
WMC Weighted Methods per Class.
WR Write/Read.

List of Tables

List of Figures

List of Listings

1 Introduction

In 1996, during the inaugural flight of the Ariane 5 rocket, a data conversion error resulted in an unexpected out-of-range value which set off a sequence of events that culminated in the rocket self-destructing less than 40 seconds after launch [93]. The direct cost of the disaster was estimated at $370 million [41]. Other examples of expensive, and sometimes lethal, software defects are legion. Between 1985 and 1987, a software bug caused a Therac-25 radiation therapy unit to fatally overdose up to six patients [68]. A unit conversion error in the Mars Climate Orbiter resulted in the loss of the $125 million spacecraft [94]. A glitch in a new stock trading algorithm caused the computers of the Knight Capital Group trading firm to go on a 45 minute buying spree that resulted in a loss of $440 million for the company [19].

In order to prevent these costly software defects, developers spend a significant amount of their time, more than half of the total development time according to some studies [54, 104], testing and debugging their software. An attractive option for reducing the time spent manually testing code, while even improving the test coverage of the tested code, is to rely on automated testing techniques such as *fuzz testers*. Fuzz testers autonomously test code by executing the program while generating, or *fuzzing*, user inputs and system inputs. By generating a diverse set of inputs, fuzzers can explore how the program behaves under various circumstances.

There are a wide variety of approaches for fuzzing program inputs, ranging from black-box fuzzers, which generate inputs without analysing the source code of the program under test [74], to white-box fuzzers, which employ sophisticated analyses to generate inputs [85]. A prominent example of whitebox fuzzing is *concolic testing*, which collects symbolic constraints over program inputs to construct path conditions that precisely model which classes of input values steer execution along specific program paths. Because of their systematic nature and their ability to find bugs in even hard-to-reach parts of the program, concolic testers have been employed in industry [52, 89, 64]. In addition to their succes in industry, the application of concolic testing to sequential, single-process applications has been well-studied in academic literature.

However, in recent years, application development has moved away from traditional, single-process desktop applications and shifted towards the development of web applications. Advantages of this type of applications include increased accessibility, as they can be accessed from both desktop computers or mobile devices, and improved ease of use, as these applications do not need to be installed or updated by their users. Examples of such applications include collaborative text editors, spreadsheets, and drawing applications. We define *full-stack JavaScript web applications* as web applications where both the client and the server have been implemented in JavaScript, and which communicate via technologies that are accessed through JavaScript, such as Websockets or libraries for HTTP communication.

Automated testing of such web applications is complicated by the unique set of challenges that arise for these applications, including the event-driven and distributed nature of the client and server processes. In the face of these challenges, creating a concolic tester for these applications that is capable of fully exploring the program, even across process boundaries, while remaining precise and avoiding false positive errors is non-trivial. **This thesis investigates how a 'precise', i.e., without reporting false positive errors, concolic tester for full-stack JavaScript web applications can be built.**

1.1 Full-stack JavaScript Web Applications

Consider the WHITEBOARD[1] collaborative drawing application depicted in Figure 1.1, which is an example of a full-stack JavaScript web application. This application allows multiple clients to connect to one common server to support creating collaborative drawings. Each new line segment is sent to the server, which in turn shares the drawing with the other clients. Users can pick a colour from the palette.

This example highlights how full-stack JavaScript web applications combine an interesting set of features. Like traditional web applications, they consist of a client process and a server process that communicate with each other to serve

[1]Source code available at `https://github.com/devansvd/whiteboard-socketio`

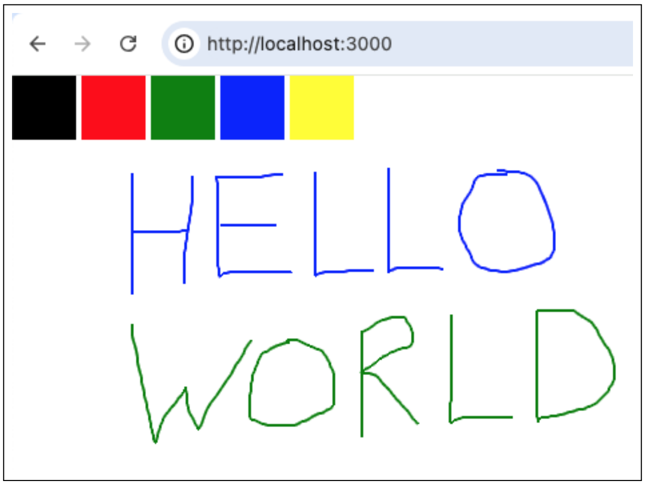


Figure 1.1: A collaborative drawing application as an example of a full-stack JavaScript web application.

responses to a user's interactions. Both processes are event-driven. The WHITEBOARD server registers a message handler to listen for incoming drawing messages from its clients. A client registers event handlers for each of the coloured boxes so that, when the user clicks one of them, new lines are drawn in the corresponding colour. It registers event handlers for mouse actions on the canvas to construct invidual line segments. A client also registers a message handler to listen for incoming messages from the server, which shares other clients' drawings via these messages. A large part of the code of both the clients and the server is hence hidden behind event handlers and message handlers, to be executed only when the corresponding event or message has been activated.

Execution of the client and server is heavily dependent on each other. Whenever a client has finished drawing a new line, it serialises the line segments and sends them to the server. The server receives the message and transmits the new line segment information to the other clients. These other clients in turn receive this information and invoke some functionality for visualising the line segments on their canvas. The execution of one process, i.e., the first client drawing a new line, hence affects the execution of not only the server process, but also the other client processes of the application.

The number of processes is furthermore not necessarily fixed, as an application may consist of a variable number of client and server instances. An essentially unlimited number of clients can connect to the same server in the WHITEBOARD application. The server may also scale up to multiple instances to reduce the load.

Note that this application does not make use of web workers, which allow for multithreading of JavaScript code on the same client process. The inclusion of web workers stands orthogonal to our definition of full-stack JavaScript web applications. Web workers allow for executing code in parallel, but do not, by themselves, make it possible to introduce new functionalities in the web application.

1.1.1 Concolic Testing of Full-stack JavaScript Web Applications

Like sequential, single-process applications, developers of full-stack JavaScript web applications may wish to have automated testers, such as concolic testers, test their code. However, the unique set of characteristics of these applications makes it non-trivial to have a concolic tester validate their correctness, as these characteristics give rise to several unique challenges:

1. The event-driven nature of the client and server processes forces a concolic tester for these applications to craft elaborate event and message sequences in order to fully cover the corresponding process's event and message handlers.

2. These applications consist of multiple communicating processes. Because of the interconnected nature of these processes, the execution of one process may affect that of another in unexpected ways. This forces the tester to not only consider a process in isolation, but to also consider the interplay between processes.

3. These applications often do not have a fixed number of instances of the client and server. Some bugs may only manifest themselves when a specific number of processes are connected. The concolic tester must therefore be capable of spawning a variable number of client and server processes.

4. JavaScript is notoriously difficult to test because of its dynamic nature, including dynamic code evaluation and a dynamic typing system, and its permissive attitude towards program faults. A concolic tester for these applications must therefore be capable of handling JavaScript's idiosyncrasies.

Apart from the difficulty involved in solving these individual challenges, some challenges conspire to make concolic testing in general even more difficult. Concolic testers systematically explore programs by traversing the program's input space and observing how different sets of inputs affect the execution of the program. For example, some inputs in the input space may cause the predicate of an `if` statement to evaluate to `true`, while others may cause it to evaluate to `false`. Increasing the size of the input space, by including *events* in the case of event-driven programs, or by considering inputs from multiple processes, results in an exponential increase in the time required by a concolic tester to exhaustively test a program. This concept is referred to as the problem of *state explosion*.

Finally, concolic testers aim to be *sound*: every bug that they report should be a bug that can actually manifest itself when executing the program [141]. This necessitates that concolic testers precisely model the behaviour of the application in function of the program inputs that are generated by the tester. A concolic tester for full-stack JavaScript web applications must therefore remain capable of maintaining precision in the face of these challenges.

1.1.2 Problem Statement

We categorise the various problems that concolic testers for full-stack JavaScript web applications must overcome as: **testing event-driven websites**, performing **inter-process testing**, and **preventing state explosion**.

Testing Event-driven Websites

Websites are no longer static pages consisting of just HTML. They are sophisticated, event-driven programs, created via a mix of HTML, CSS, and JavaScript. The JavaScript allows for dynamically updating the content of the website and enables communication with the server by other means than just HTTP requests, such as Websockets. The content of the page itself may affect the execution of the JavaScript code of the website. Users may be able to enter values through input fields on the page. If these values are read by the JavaScript code, they can in turn affect the execution of the code.

As users interact with full-stack JavaScript web applications via a website running on a specific client, a concolic tester for these applications must therefore necessarily be capable of testing event-driven websites as well. A simplified view of these websites is that they are essentially an event-driven JavaScript web application that interacts with a website's Document Object Model (DOM). Although many concolic testers for such applications have been presented [111, 123, 106, 72], it remains difficult to build a concolic tester that is capable of handling JavaScript's dynamic nature, can model user inputs that originate in the DOM, and is capable of generating user and system events.

Inter-process Testing

We observe that existing concolic testers for JavaScript are insufficient to test full-stack JavaScript web applications, because they are mostly *intra-process* testers: they test only a single JavaScript process, and their execution paths do not cross process boundaries. This approach does not suffice for full-stack JavaScript web applications, as these applications consist of multiple, communicating processes. Intra-process testers test individual processes in isolation. Although they may be capable of fully covering the process under test, i.e., testing all of its possible behaviours, testing a single, isolated process is much less precise than testing the process in a composition with other processes.

Intra-process testers cannot distinguish between the parts of the code of the process under test that are *reachable* in a given composition of processes, and the parts of the code that are not reachable. For example, an intra-process tester for a server process may aim to test the execution of a message handler, even to the point where it significantly exhausts its test budget in attempting to completely cover the handler, without realising that in the actual full-stack JavaScript web application, no client will ever send this message in the first place.

Furthermore, when testing a handler in the server process that receives a message from a client process with some message payload, an intra-process tester for the server cannot determine *how* the client computed the values that were sent as the message payload to the server. The intra-process tester therefore cannot construct fully precise constraints over the execution of the message handler as it does not know from where the payload originated, and hence which execution paths through the handler are possible.

A solution to this problem involves *inter-process* testing, which we define in this thesis as *testing the composition of all processes in a full-stack JavaScript web application as a whole, while observing their communication, and having their execu-*

tion paths cross process boundaries. This then leaves the problem of constructing a concolic tester that can spawn multiple processes, observe their communication, and model cross-process execution paths.

Preventing State Explosion

The final problem concerns how concolic testing can avoid the state explosion problem when testing full-stack JavaScript web applications. When testing an application, the number of unique *states* that the application may assume generally grows exponentially in function of the number of branch conditions, such as `if` statements, that are encountered while executing the program. Every branch condition may lead the execution along a different program path, where e.g., program variables may be assigned values that slightly differ from those along different paths. The amount of time required by a concolic tester that aims to exhaustively cover all possible states in an application therefore grows exponentially in function of the complexity of the application.

This state explosion problem manifests itself already when testing relatively simple, sequential, single-process applications. The problem grows even worse when testing full-stack JavaScript web applications as these applications present additional sources of complexity. Their event-driven nature forces a concolic tester to also consider execution paths that flow along event handlers and message handlers, thereby increasing the number of potential states. The distribution of an application over multiple processes also increases the number of states that the global application may assume, as the execution paths of the various individual processes compose to create an even larger set of execution paths.

State merging [13], which merges similar program states together, presents itself as a possible solution to this problem. However, state merging has not yet been defined for concolic testing of full-stack JavaScript web applications. It is therefore not clear how state merging can be incorporated into a concolic tester for these applications.

1.2 Overview of the Approach

This dissertation presents *StackFul*, an inter-process concolic tester for full-stack JavaScript web applications that applies state merging to event-driven programs in order to prevent the state explosion problem from arising. We develop StackFul through the following intermediate steps:

1. We identify challenges for performing automated testing on full-stack JavaScript web applications (Section 2.3) and distil these challenges into seven criteria (Section 2.4) that automated testers for these applications should satisfy.

2. We build an intra-process concolic tester, named *$\text{StackFul}_{\text{INTRA}}$*, that is capable of testing single-process JavaScript programs. $\text{StackFul}_{\text{INTRA}}$ uses *shadow execution* to execute a program while simultaneously collecting symbolic constraints that precisely model the program path followed by the execution (Sections 4.1–4.3).

3. We extend $\text{StackFul}_{\text{INTRA}}$ to make it capable of testing event-driven programs (Section 4.4). With the capability of testing event-driven, single-process JavaScript applications, we evaluate $\text{StackFul}_{\text{INTRA}}$ on the set of seven criteria (Section 4.6).

4. We build on $\text{StackFul}_{\text{INTRA}}$ to develop *$\text{StackFul}_{\text{INTER}}$*, an inter-process concolic tester for full-stack JavaScript web applications. $\text{StackFul}_{\text{INTER}}$ tests the composition of all client and server processes in an application as a whole, while observing their communication, so that it can construct precise constraints over the application's execution paths (Sections 5.1–5.2). We motivate the need for inter-process testing by using $\text{StackFul}_{\text{INTER}}$ to distinguish between high-priority and low-priority server errors, where we define a high-priority server-side error in a particular client-server configuration as an error that is reachable from *any* client of this application. Low-priority server-side errors are defined as server-side errors that are *not* reachable from any of the clients in the client-server configuration of the application (Section 5.3.1).

5. We formalise both $\text{StackFul}_{\text{INTRA}}$ and $\text{StackFul}_{\text{INTER}}$ to precisely capture the essential aspects of intra-process (Section 4.5.1) and inter-process (Section 5.4) concolic testing.

6. Concolic testing for sequential, single-process applications gives rise to the state explosion problem. This problem is exacerbated when concolic testing is performed on event-driven code, and when execution paths are made longer by having them cross process boundaries. We therefore incorporate state merging into StackFul$_{\text{INTER}}$, naming the resulting tester *StackFul*, and investigate how it can be applied to concolic testing of event-driven code (Section 6.3), so it can be used to test full-stack JavaScript web applications.

We have made the source code of StackFul publicly available at `https://github.com/softwarelanguageslab/StackFul`

1.3 Contributions

This dissertation presents contributions along three main axes: i) **identifying the challenges in automated testing of full-stack JavaScript web applications**, ii) the concept of **inter-process concolic testing**, and iii) **state merging**.

Specifically, we contribute to the first axis:

- We identify four challenges in performing automated testing of full-stack JavaScript web applications and distil these into seven criteria that automated testers for these applications should satisfy.

We present two contributions, supported by a publication in an international, peer-reviewed journal [129], for the second axis:

- We define the concept of inter-process concolic testing as concolic testing where a composition of multiple instances of the client and server processes are tested as a whole, while observing their communication. Symbolic path constraints are collected along execution paths that cross process boundaries. We implement inter-process testing in the StackFul$_{\text{INTER}}$ concolic tester.

- We implement a use case in StackFul$_{\text{INTER}}$, in which we employ the increased precision offered by inter-process testing to distinguish between high-priority and low-priority server errors in full-stack JavaScript web applications. We define a high-priority server error as a server error that is reachable by a client in a specific composition of client-server processes, whereas a low-priority server error is an error that is only theoretically reachable.

For the third axis, we contribute:

- We present how state merging can be applied in concolic testing of event-driven programs. We present StackFul, which was built on top of StackFul$_{\text{Inter}}$ but incorporates this form of state merging.

1.4 Publications

In this section, we list work that has been conducted over the course of this PhD and which has been published at several international, peer-reviewed venues. We distinguish between publications that are directly related to the work presented in this dissertation and publications that are related only indirectly.

1.4.1 Supporting Publications

We list and summarise the publications directly related to this thesis below:

- Vandercammen, Maarten, Christophe, Laurent, De Meuter, Wolfgang, and De Roover, Coen. (2018). Concolic Testing of Full-stack JavaScript Applications. In *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop*, Delft, the Netherlands, pages 38–42. (Workshop paper)

This workshop paper proposes the idea of performing concolic testing on all processes of a full-stack JavaScript web applications. The paper motivates this idea with potential improvements in the precision of such a tester compared to traditional testers which test processes in isolation from each other.

- Vandercammen, Maarten, Christophe, Laurent, Di Nucci, Dario, De Meuter, Wolfgang, and De Roover, Coen (2020). Prioritising Server Bugs via Inter-process Concolic Testing. *The Art, Science, and Engineering of Programming*, **5**(2), 5.

This paper builds on the idea of concolic testing for full-stack JavaScript web applications by introducing the concept of *inter-process* concolic testing, and describing the implementation of StackFul$_{\text{INTRA}}$ and StackFul$_{\text{INTER}}$. The paper also presents a use case for employing inter-process testing to distinguish between high-priority and low-priority server-side errors (cf. Chapter 5), and evaluates this approach on a suite of full-stack JavaScript web applications.

- Vandercammen, Maarten and De Roover, Coen. State Merging for Concolic Testing of Event-driven Applications. Under review at the *Journal of Systems and Software.*

This paper, under review at the time of the publication of this dissertation, presents a novel approach for incorporating state merging into concolic testing of event-driven applications (cf. Chapter 6).

This thesis also directly contributed to the successful completion of several master and bachelor theses.

1.4.2 Other Publications

We also present publications that are not directly related to this thesis, with the aim of describing our academic realisations outside of the subject of inter-process concolic testing of full-stack JavaScript web applications. Although this work has no direct relation with the topic of this thesis, it indirectly influenced the approach taken in this thesis by its focus on static and dynamic analysis and automated testing.

A large part of our work was focused on static analyses performed at run time, i.e., *online static analysis.* A key benefit of online static analysis is that the analysis can incorporate concrete information about the execution of the program to improve the precision of the analysis. We first investigated whether online static analyses could be applied to the domain of Just-In-Time (JIT) compilation of dynamic languages, to improve the optimisations performed on the JIT-compiled code. The idea was to perform an extremely lightweight online static analysis on the entire program after collecting the code to be JIT-compiled. The increased precision about the execution of the program outside the JIT code snippet would enable the compiler to further optimise the JIT-compiled code, for example by extending the scope of constant propagation optimisations or by eliminating code that computes values which are not used outside of the JIT-compiled code. This work resulted in the following publications:

- Vandercammen, Maarten, Nicolay, Jens, Marr, Stefan, De Koster, Joeri, D'Hondt, Theo, and De Roover, Coen. (2015). A Formal Foundation for Trace-Based JIT Compilers. In *Proceedings of the 13th International Workshop on Dynamic Analysis*, WODA, Pittsburgh, PA, USA, pages 25–30. (Workshop paper)

- Vandercammen, Maarten, Marr, Stefan, and De Roover, Coen (2018). A Flexible Framework for Studying Trace-based Just-In-Time Compilation. *Computer Languages, Systems & Structures*. **51**, pages 22–47.

- Vandercammen, Maarten, Stiévenart, Quentin, De Meuter, Wolfgang, and De Roover, Coen. (2015). STRAF: A Scala Framework for Experiments in Trace-based JIT compilation. In *Grand Timely Topics in Software Engineering - International Summer School*, GTTSE, Braga, Portugal, pages 223–234. (Workshop paper)

- Vandercammen, Maarten and De Roover, Coen. (2016). Improving Trace-based JIT Optimisation Using Whole-Program Information. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, VMIL, Amsterdam, The Netherlands, pages 16–23. (Workshop paper)

The first two publications present a formal model of trace-based JIT compilation, which allows for easily lifting the concrete information observed by the JIT compiler to the abstract domain employed abstract interpreters [37]. The third publication presents a framework for implementing an extensible trace-based JIT compiler, which served as the research vehicle in which this research was conducted. The fourth publication describes our investigation into online static analysis for improving trace-based JIT optimisations, and presents some cases where optimisations involving constant propagation are improved because of the analysis.

However, the number of scenarios in which online static analysis significantly improves the performance of the compiled code are limited. We therefore investigated whether online static analysis can be combined with concolic testing, to steer the tester away from parts of the application that can be proven safe at run time. This work resulted in the following publication:

- Vandercammen, Maarten and De Roover, Coen. (2017). Employing Runtime Static Analysis to Improve Concolic Execution. In *Proceedings of the 16th edition of the Belgian-Netherlands software evolution symposium*, Antwerp, Belgium, pages 26–29. (Workshop paper)

1.5 Outline of the Dissertation

The remainder of this dissertation is structured as follows:

Chapter 2: Automated Testing of Full-stack JavaScript Web Applications We provide background material on the problem and solution domains of this dissertation: full-stack JavaScript web applications and concolic testing. We describe the challenges that arise when applying concolic testing to full-stack JavaScript web applications, and we distil these challenges into seven criteria that testers for these applications should satisfy.

Chapter 3: State of the Art in Concolic Testing We contextualise our tester by describing the current state of the art in automated testing of these applications. We focus on automated testers for distributed systems, event-driven applications, JavaScript applications, and web servers. We also give an overview of generic optimisation techniques that can be applied to these testers. We conclude by evaluating selected testers on the seven criteria identified in the previous chapter.

Chapter 4: A Foundation of Intra-process Concolic Testing We introduce *StackFul*$_{\textit{INTRA}}$, an intra-process concolic tester for single-process JavaScript applications. We extend concolic testing of sequential JavaScript code to testing of event-driven applications. We describe how StackFul$_{\text{INTRA}}$ simultaneously executes a program concretely and symbolically through shadow execution, and formally detail it.

Chapter 5: Inter-process Concolic Testing We build on the concolic tester introduced in the previous chapter and introduce *StackFul*$_{\textit{INTER}}$, an inter-process concolic tester. StackFul$_{\text{INTER}}$ tests the composition of all instances of clients and servers in a full-stack JavaScript web application as a whole while observing their communication in order to increase precision and avoid false positive errors. As a use case, we demonstrate how inter-process testing can be used to distinguish between high-priority and low-priority server errors in a full-stack JavaScript web application, based on whether a server-side error is reachable from a client in a particular client-server configuration or not.

Chapter 6: State Merging for Event-driven Programs We outline the state explosion problem that arises when performing concolic testing of sequential, single-process applications, and explain how it is made worse by event-driven code. We describe how state merging alleviates the state explosion problem, and we lift this technique to the domain of concolic testing of event-driven applications. We incorporate state merging into StackFul$_{\text{INTER}}$ and name the resulting tester *StackFul*.

Chapter 7: Conclusion We conclude the dissertation by revisiting the problem statement and listing potential avenues for future work.

2 Automated Testing of Full-stack JavaScript Web Applications

In this chapter, we introduce the two main concepts that are the focus of this dissertation: full-stack JavaScript web applications and techniques for automated testing, specifically concolic testing. We start by defining characteristics that distinguish full-stack JavaScript web applications from traditional applications and present a working example of an application that will be used throughout the chapter.

Section 2.2 introduces a variety of automated testing techniques, with emphasis on concolic testing. Section 2.3 combines the previous two sections by outlining the unique challenges that full-stack JavaScript web applications pose for automated testers. Automated testing of full-stack JavaScript web applications faces unique challenges that do not arise when testing non-distributed, sequential applications. These challenges relate to the heterogeneous configuration of processes in full-stack JavaScript web applications, the interconnected nature of these processes, the event-driven code employed by the clients and servers of these applications, as well as the dynamic nature of JavaScript and its permissive attitude towards program faults. Finally, Section 2.4 distils these challenges into concrete criteria that an automated tester for full-stack JavaScript web applications should adhere to in order to effectively test such applications.

2.1 Full-stack JavaScript Web Applications

In this section, we introduce and define the term *full-stack JavaScript web application.* We specify unique characteristics that distinguish these applications from more traditional applications in Section 2.1.1 and present an example implementation that highlights these characteristics in Section 2.1.2.

2.1.1 Characteristics

Traditionally, a web application is divided into two components: the frontend and the backend [1]. The frontend consists of all parts through which a user interacts with the application. These comprise the web pages with their graphical user interface elements that are accessed directly by the user. The backend includes the servers on which the application logic is running and the databases employed in the application. These technologies, as well as the glue services that make it possible for the technologies to communicate with each other, collectively form the *technology stack* of the application [121]. Each layer of the technology stack gives rise to one or more processes that are spawned, such as a server process, a web client process, a database process etc. The constituent processes of the application commonly run on entirely different machines: e.g., the frontend is executed on a user's machine, whereas the backend is executed on the organisation's machines.

Technology stacks can be customised to specific applications or follow a standard template. Examples of such templates are the LAMP stack (Linux, Apache, MySQL, PHP), the MEAN stack (MongoDB, Express, Angular, Node.js) which supports the development of JavaScript applications or the Ruby stack (relying on the Ruby-on-Rails framework[1]) [121]. As an example of such a technology stack, Figure 2.1 depicts the MEAN stack. Users interact with the application via the frontend, which consists of one or multiple web pages developed in the Angular[2] framework. The user's actions may cause the page to communicate with the backend, for example to request loading of certain data or to update the state of the application. The backend consists of the server (written in Express[3] and Node.js[4]) and the database (MongoDB[5]). The Express framework, running on Node.js, is a middleware framework. It enables specifying which actions the server must undertake when it receives an HTTP request [45]. Lastly, the server uses the MongoDB database framework for persistence.

The term *"full-stack development"* is only informally defined in current literature, but is generally taken to refer to "[technologies across] the entire depth of the application's technology stack" [24, 70, 84, 121, 79]. Full-stack development thus takes place along all layers of these technology stacks: in components situated along both the frontend and the backend, as well as the infrastructure through

[1]`https://rubyonrails.org/`
[2]`https://angular.io/`
[3]`https://expressjs.com/`
[4]`https://nodejs.org/en`
[5]`https://www.mongodb.com/`

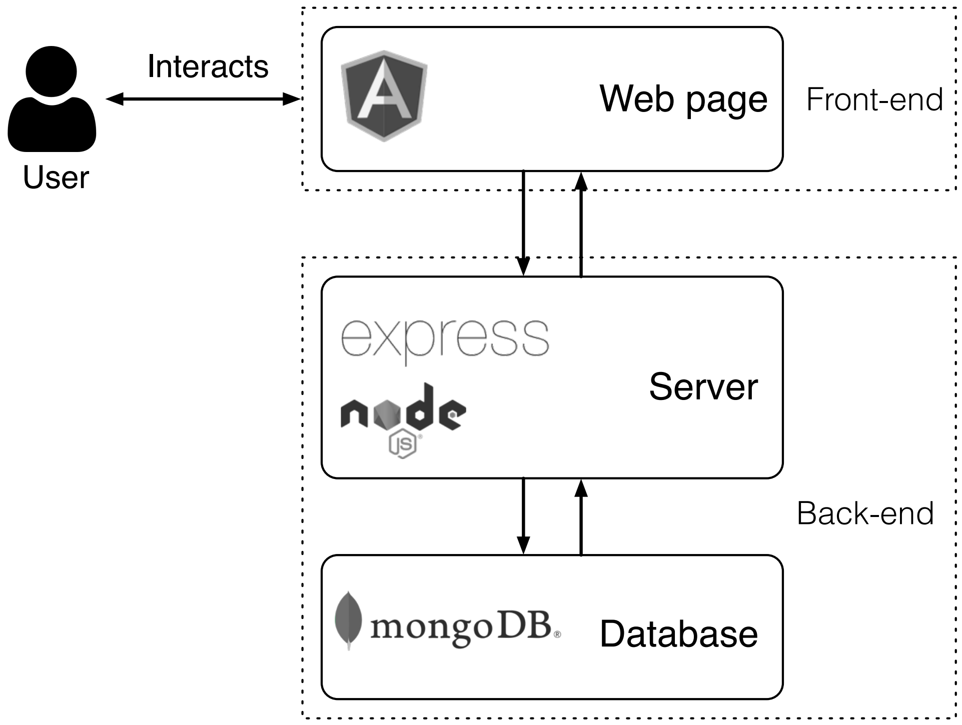


Figure 2.1: The MEAN (MongoDB, Express, Angular, Node.js) technology stack.

which these technologies communicate with each other. Full-stack developers are expected to contribute to the development of all layers of the application's technology stack and therefore require the skill sets necessary for working with these technologies. *Full-stack JavaScript web applications* are the applications built by combining several of these technologies, where the frontend consists of one or more web pages that communicate with a backend which possibly runs on another machine, and where both the frontend and the backend have been implemented in JavaScript. In short, we define full-stack JavaScript web applications as *web applications where both the client and the server have been implemented in JavaScript, and which communicate via technologies that are accessed through JavaScript, such as Websockets or libraries for HTTP communication.*

2.1.2 Example Application

We present an example of a full-stack JavaScript web application, Calculator, which exhibits the characteristics of these types of applications that were outlined in the previous section. This application, of which the client-side user-interface is depicted in Figure 2.2, is an online calculator that allows users to write arithmetic expressions by entering numbers into the input field or by clicking the buttons. When the user clicks the button labelled with the equality sign, the expression is sent to the server. The server receives this expression, parses it, performs the arithmetic computation, and sends the result back to the client so it can be shown to the user.

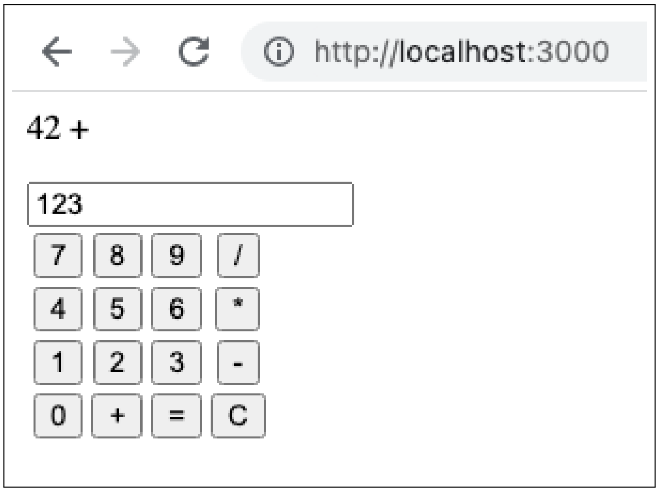


Figure 2.2: A screenshot of the Calculator application's frontend.

The technology stack of Calculator is depicted in Figure 2.3. Like other full-stack JavaScript web applications, it consists of a frontend and a backend. The frontend consists of a web page, used to give the user access to the application via the user-interface depicted in Figure 2.2, which is implemented in plain HTML and JavaScript, without the use of any additional frameworks. The backend consists of a server, which runs Node.js and uses Express to serve the frontend web page when a user connects to the application. As with other full-stack JavaScript web applications, the frontend and backend can run on different machines. Furthermore, several instances of the frontend may be spawned, with all of them connected to the same backend instance. Since the application has no need for persisting data, Calculator does not use a database.

Bidirectional sockets, provided through the Socket.IO library[6], serve as the glue between the frontend and backend. These sockets enable frontend and backend processes to communicate with each other, by allowing for the transfer of primitive JavaScript values (e.g., strings, numbers, and booleans) and serialisable objects.

[6]`https://socket.io/`

The remainder of this section provides an overview of the frontend and backend of Calculator, demonstrating the characteristics of full-stack JavaScript web applications.

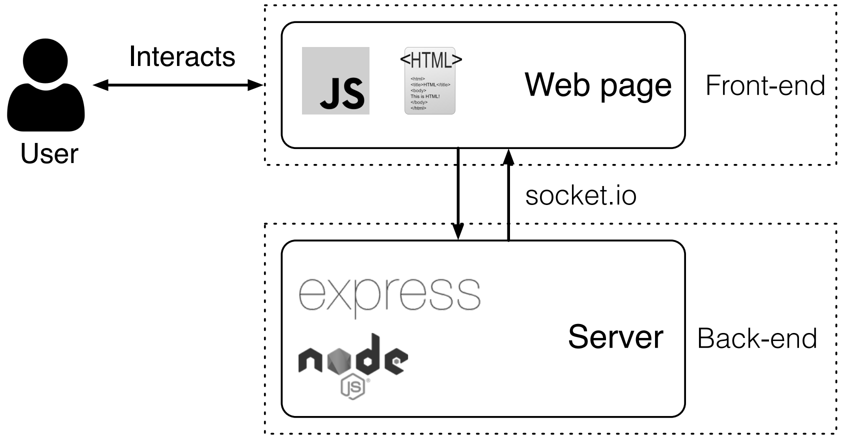


Figure 2.3: The technology stack of the Calculator application.

Application Frontend

Listing 2.1 depicts an extract of the frontend implementation. The client calls function `io` (line 1) for a connection to the server through a bidirectional channel from the Socket.IO library. Rather than defining event handlers that dictate the appropriate action for clicking each button in the HTML document of the web page, these handlers are registered dynamically via JavaScript. The client registers a separate handler for a mouse click on each individual button (lines 2 to 8). Note that the event handlers for most buttons have been elided from the listing. Importantly, the event handler for the button labelled "`=`" calls the `compute` function (line 7). The client also registers an event handler for key presses on the text input field (line 9). When the user enters numbers or text into this field, the user input is parsed by the elided function `keyPressed`.

The client represents the arithmetic expression as an object `input` (line 12) containing three fields `left`, `op`, and `right`. Function `compute` checks whether the expression that was entered is a valid arithmetic expression with a recognised arithmetic operator (line 14). If the check fails, the function shows an appropriate error message to the user (line 16). Otherwise, the function sends `input` to the server through the socket (line 18). Socket.IO's `emit` method automatically serialises the `input` object as a JSON string. Finally, the client registers a callback for messages from the server (line 22). After the server completes the computation, it sends the result to the client through the `result` parameter of the callback. Upon receiving such a message, the client shows the result to the user (line 24).

```
1  const socket = io(); // Connect socket with the server
2  const buttonZero = document.getElementById("Button0");
3  const buttonPlus = document.getElementById("Button+");
4  const buttonEquals = document.getElementById("Button=");
5  buttonZero.addEventListener("click", (evt) => clickDigit(0));
6  buttonPlus.addEventListener("click", (evt) => clickOperator("+"));
7  buttonEquals.addEventListener("click", (evt) => compute());
8  ... // Register event handlers for other buttons
9  const inputField = document.getElementById("input");
10 inputField.addEventListener("keypress", (evt) => keyPressed(evt););
11
12 const input = {left: 0, op: "", right: 0}; // Arithmetic exp
13 function compute() {
14   if (! isValidExpression(input)) {
15     // Warn user of invalid expression
16     resultElement.innerHTML = "Expression is invalid";
17   } else {
18     // Send the expression to the server
19     socket.emit("compute", input);
20   }
21 }
22 socket.on("result", function (result) {
23   // Receive computation result from server
24   resultElement.innerHTML = result; // Show the result
25 });
```

Listing 2.1: Part of the JavaScript frontend code of the CALCULATOR application.

The frontend furthermore uses HTML to define the domain object model (DOM) of the web page. An extract of the HTML code is depicted in Listing 2.2. The HTML extract specifies a paragraph element where the evaluated result of the arithmetic expression will be shown (line 8), the text input field where the user can enter a number (line 9), and three of the buttons that users may click to form the arithmetic expression (lines 13–16).

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Calculator</title>
    </head>

  <body>
    <p id="Result"></p>
    <input type="text" id="input"></input>
       <table>
         <tr>
           ...
           <th> <button type="button" id="Button1">1</button> </th>
           <th> <button type="button" id="Button2">2</button> </th>
           <th> <button type="button" id="Button3">3</button> </th>
           <th> <button type="button" id="Button-">-</button> </th>
         </tr>
         ...
```

Listing 2.2: Part of the HTML frontend code of the CALCULATOR application.

Application Backend

Listing 2.3 depicts an extract of CALCULATOR's backend's code. The code first creates an Express application (lines 1–2), and then sets up an `http` server while instructing the server to use the Express application as a middleware stack (line 3). The Express application is configured to serve the static HTML page (cf. Listing 2.2) that is located in the `public` folder of the server's file system (line 5). The server is furthermore configured to open port 3000, although no action is taken when a client connects to this port (line 29).

Orthogonal to this server setup, the code also configures (line 4) a Socket.IO channel that listens for incoming connections from clients (line 6). When a new client connects, the corresponding callback is triggered (lines 6–28) with the socket through which the client is connected as an argument. The server registers a message handler (line 8) on each socket to listen for `compute` messages coming from the corresponding client. When the client sends such a message, the arithmetic expression is automatically deserialised from a JSON string into a JavaScript object. The server retrieves from this object the left and right operand, as well as the operator (lines 10–12). The result is computed from these three elements (lines 14–24) and sent back to the client via its socket (line 26). Importantly, the server throws an error when it detects a division by zero (line 20) or when it does not recognise the operator to be applied (line 23).

```
 1 const express = require("express");
 2 const app = express();
 3 const http = require("http").createServer(app);
 4 const io = require("socket.io")(http);
 5 app.use(express.static(__dirname + "/public"));
 6 io.on("connection", function(socket) {
 7   // A new client has connected
 8   socket.on("compute", function (input) {
 9     // Receive input from client
10     const left = input.left;
11     const right = input.right;
12     const op = input.op;
13     let result;
14     switch (op) {
15       case "+": result = left + right; break;
16       case "-": result = left - right; break;
17       case "*": result = left * right; break;
18       case "/": result =
19         if (right === 0) {
20           throw new Error("Dividing by zero");
21         }
22         result = left / right; break;
23       default: throw new Error("Unknown operator");
24     }
25     // Send the result to the client
26     socket.emit("result", result);
27   }
28 });
29 http.listen(3000, function () {});
```

Listing 2.3: Part of the backend code of the CALCULATOR application.

2.2 Automated Testing Techniques

The previous section introduced full-stack JavaScript web applications. In this section, we introduce various techniques for automated testing. Later, in Chapter 3, we give a more detailed overview of the state of the art on these automated testing techniques. In Section 2.3, we discuss the unique challenges brought about by full-stack JavaScript web applications for automated testing.

As a matter of terminology, we distinguish between *"test automation"* and *"automated testing"*. We define "test automation" as technologies that help automate the execution of tests that were specified manually. Examples of test automation technologies include frameworks such as JUnit[7] or Jest[8], which enable developers to write unit tests that can be executed repeatedly and automatically by the framework. Test automation may also use capture-replay techniques to e.g., test the program's GUI. Apart from unit testing, test automation can also be used to perform end-to-end testing, performance testing, regression testing, functional requirement testing, integration testing, etc. [44]. Benefits to test automation, as perceived by industry practitioners, include reduced developers' time spent on testing, improved test coverage, reusability of tests, and increased quality and reliability of the final software product [100].

We use "automated testing" to refer to using software tools, called *"automated testers"*, that autonomously execute a program, or parts thereof, under controlled circumstances while reporting erroneous behaviour. Automated testers do not follow manually specified tests, but automatically generate test inputs to drive the execution of the program under test. In this section, we focus on automated testing where an application is autonomously executed with algorithmically generated test inputs and its behaviour or outputs monitored. These inputs may take the form of e.g., user inputs which would otherwise be provided by a user manually interacting with the application, message inputs representing communication data received by the system, inputs read from a file by the application, program inputs etc.

We introduce the following types of automated testers: *fuzz testing*, *search-based testing*, and *concolic testing*.

[7]https://junit.org/junit5/
[8]https://jestjs.io/

2.2.1 Fuzz Testing

In general, *fuzzing* refers to the practice of repeatedly executing a program while producing input that is (intentionally) malformed in order to trigger unexpected behaviour [85]. Fuzzing can be used by malicious actors to exploit security vulnerabilities in a program or to penetrate its defences [29]. Conversely, the developers of these systems have taken up fuzzing in order to pro-actively discover weaknesses and deny attackers these vulnerabilities.

Fuzzing can also be used to detect forms of erroneous behaviour other than security violations. *Fuzz testing* hence refers to the practice of using fuzzing to test the correctness of programs. It has been successfully applied in various contexts and has been used by e.g., Google [58], Facebook [4], and Microsoft [18, 51] to validate their applications.

Fuzz testing techniques can be divided into the following categories, based broadly on the amount of program analysis they leverage to test the system [29, 85]: black-box fuzzing, white-box fuzzing, or grey-box fuzzing.

Black-box Fuzzing

Black-box fuzzers do not employ any program analysis at all. The fuzzer continuously executes the program with randomly generated inputs, without consideration for how these inputs affect the execution of the system, while monitoring the outputs produced by the system. Some black-box fuzzers use an input grammar that specifies an appropriate format or structure for the input, in order to pass superficial program checks on the input [85].

Methods for generating new inputs for these fuzzers are either mutation-based or generation-based [88]. Mutation-based methods start from a suite of prepared input seeds and create new inputs by iteratively performing random mutations on these inputs. Generation-based methods employ a formal specification of the structure of the input, such as a grammar, to produce new inputs from scratch.

The main advantage of black-box fuzzing lies in its simplicity [76]. However, because these fuzzers are ignorant of the code they are executing, they may find it difficult to cover branches that expect inputs to assume very specific values. As such, black-box fuzzers often achieve only a low code coverage [52].

White-box Fuzzing

White-box fuzzers stand on the opposite side of this spectrum, as they have the entire code base of the program available for analysis. Such fuzzers may employ symbolic execution [21] to construct precise constraints of the program paths that were executed. By manipulating these paths and feeding them to a constraint solver, the fuzzer can compute inputs which will cause the execution to follow a different, unexplored path in the next fuzzing run.

In theory, this allows the tester to systematically enumerate all possible program paths and therefore to uncover all bugs that are present in the system. In practice, however, the set of potential program paths may grow infeasibly large. To manage this search space, the fuzzer has to rely on sophisticated search heuristics to select appropriate paths to explore [26]. One form of white-box fuzzing, concolic testing, will be described in more detail in Section 2.2.3.

Examples of white-box fuzzers include KLEE [23], SAGE [51, 52], and PEX [128].

Grey-box Fuzzing

Situated between these two extremes are grey-box fuzzers, which partially instrument the system in order to collect coverage information. The collected feedback helps the fuzzer increase code coverage by making it possible to select for program inputs that increase coverage. Other parts of the program may then be uncovered by further mutating those selected inputs.

Algorithms from the machine learning community have been leveraged to improve the selection process of new inputs. For example, Directed Greybox Fuzzing [16] uses a form of simulated annealing in which the fuzzer first launches an exploration phase in which it aims to maximize code coverage, followed by an exploitation phase in which it further manipulates successful program inputs to discover bugs. Yet other grey-box fuzzers use a Markov chain approach [17]. A prominent example of grey-box fuzzing is *search-based software testing.*

2.2.2 Search-based Software Testing

In general, *search-based software engineering* aims to solve computational problems related to the field of software engineering via various search techniques, such as genetic search algorithms [119], simulated annealing [134], or hill climbing [56]. Search-based software testing therefore represents the application of such search techniques to the domain of software testing [87, 57, 63]. Search-based software testing has been used to find errors in a program under test [3], to reproduce crashes [40], and to automatically fix simple errors [86].

As an example of grey-box fuzzing, search-based software testers employ a lightweight form of program instrumentation to e.g., measure code coverage while autonomously executing an application with generated program inputs. The choice of which inputs to execute the program with is made by the search technique, which, informed by run-time data collected by the instrumentation, attempts to select inputs that cause the test run to accomplish certain test objectives, such as maximising code coverage, covering a specific target, or verifying non-functional properties [2].

Search-based testers rely on a *fitness function* which expresses how close a particular test run under a given set of program inputs came to satisfying the test objective. The choice of which fitness function should be used therefore naturally depends on the type of test objective that the search-based tester must accomplish. Examples of fitness functions range from a branch coverage metric [48], which will direct the tester to maximise test coverage, or *approach level* and *branch distance* functions, which express how close the test run came to covering a target statement [87].

2.2.3 Concolic Testing

Concolic testing (the term being a combination of "**conc**rete and symb**olic** execution") is a form of white-box fuzzing that tests a program over multiple test runs. Concolic testing uses symbolic execution to collect so-called *path conditions* or *path constraints*. A path constraint describes the program conditions, expressed in the form of *symbolic expressions*, that must hold in order to follow a particular path through the program or code snippet under test.

By systematically negating these path constraints and using a satisfiability-modulo-theories (SMT) solver to compute solutions for program inputs that cause the negated constraint to become satisfied, the concolic tester can systematically exercise a new program path in each test run [13, 30]. In order to collect these path constraints, the concolic tester performs *concrete* and *symbolic* execution simultaneously. On the one hand the tester steers the *concrete* execution of

Table 2.1: Summary of the three test runs performed during concolic testing of the code in Listing 2.4.

Test run	Path constraint	x	y
1	$2y_0 \neq x_0$	3	5
2	$2y_0 = x_0 \wedge x_0 \leq y_0 + 10$	2	1
3	$2y_0 = x_0 \wedge x_0 > y_0 + 10$	30	15

the program along an extension of a previously-explored path and reports newly-encountered errors. On the other hand the *symbolic* execution gathers symbolic constraints over existing and newly-encountered non-deterministic variables (e.g., random values), user inputs, and program inputs that steer the program's execution so that subsequent test runs may explore different program paths.

We illustrate the workings of a concolic tester via the JavaScript example depicted in Figure 2.4. Lines 14 and 15 assign random values to the variables `x` and `y`. In general, an expression that may evaluate to different concrete values over multiple test runs of the program, such as an expression producing a random value, reading a user or program input, performing an IO read, etc., is deemed a non-deterministic expression. A non-deterministic expression symbolically evaluates to a new symbolic input variable. The tester therefore symbolically represents the results of the calls to `randomInt` as the symbolic input variables x_0 and y_0 respectively, which are then assigned to variables `x` and `y`.

Suppose that in the first test run, the concolic tester randomly assigns the values `3` to `x` and `5` to `y`. These values cause the branch condition on line 7 to be false, and the program terminates without errors. Simultaneously to this concrete execution, the tester also collects the symbolic representation of the conditional expression that was encountered on line 7 in the form of a so-called *path constraint*, i.e., $2y \neq x_0$. After completing this run, the tester attempts to explore another path, such as the path leading to the `if` statement on line 8. To this end, the tester negates the path constraint and feeds the resulting constraint $2y_0 = x_0$ to an SMT solver, which finds a solution that assigns e.g., 2 to x_0 and 1 to y_0. 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 8, then takes the `else` branch, and the program terminates again without errors. In the meantime, the symbolic execution gathered the path constraint $2y_0 = x_0 \wedge x_0 \leq y_0 + 10$. The tester negates the last element of this path constraint and feeds the resulting path constraint $2y_0 = x_0 \wedge x_0 > y_0 + 10$ into the SMT solver, which finds e.g., the

```javascript
function twice(v){
  return v * 2;
}

function f(x, y){
  let z = twice(y);
  if(z === x) {
    if(x > y + 10) {
      throw new Error();
    }
  }
}

let x = randomInt();
let y = randomInt();
f(x, y);
```

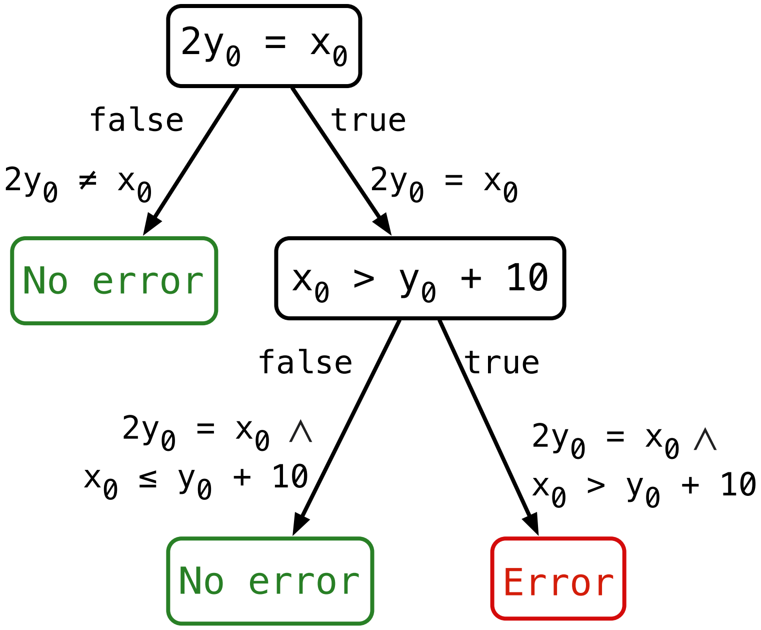


Figure 2.4: JavaScript program and its corresponding symbolic execution tree, based on Figure 1 of [21].

values 30 and 15 for x and y as a solution. A new test run is started with these values and the concrete execution reaches the error on line 9, which is reported by the concolic tester. As no new branches were encountered by the tester during this last run, the tester deduces that it has explored all feasible program paths and terminates. In practice, for realistic programs with a (near-)infinite number of program paths, the testing phase is terminated either upon exceeding a given time or search budget or when the desired level of code coverage has been reached. The three path constraints that were found can be collected in a symbolic execution tree that represents all possible executions of the program, as shown in Figure 2.4. A summary of the three concolic execution runs is presented in Table 2.1.

A concolic tester is often divided into a *test executor* and a *test selector* [62]. The test executor is the component that executes the program concretely and symbolically and that collects the path constraint over the course of a test run. The test selector constructs the symbolic execution tree from these path constraints over multiple test runs. Between test runs, it selects which path should be explored by the test executor in the subsequent run.

An important advantage of concolic testing when compared to other types of automated testers is the fact that concolic testing explores a program *systematically*. On the conditions that i) the entire code base of the application is available for symbolic analysis, ii) the concolic tester can employ an SMT solver that is capable of reasoning about the symbolic constraints with which it is presented, and iii) the tester is given a sufficiently large time budget, a concolic tester is guaranteed to explore a given, finite, satisfiable program path in a finite amount of time. For this reason, Chapter 3 details a number of state-of-the-art concolic testers for various types of applications and presents an overview of techniques for rendering concolic testing of these applications more effective.

The second constraint listed here deserves particular consideration, as SMT solvers may be incapable of computing solutions for constraints that feature complex operations, such as string comparisons or regex checks [21]. Additionally, if a value was produced in a part of the application of which the source code is not available, such as a binary library, the concolic tester will be wholly incapable of providing a symbolic representation for this value. In those cases, the concolic tester can use *concretisation*: the complex or missing symbolic expression is replaced by the concrete value that was observed during the program's execution. Although this necessarily loses accuracy, as the tester is no longer capable of precisely modelling the constraints on program inputs that cause execution to flow along a path featuring these obstacles, it does allow the concolic tester to continue making progress.

2.3 Challenges in Automated Testing of Full-stack JavaScript Web Applications

Having introduced both full-stack JavaScript web applications and automated test input generation techniques, this section describes the unique challenges posed by full-stack JavaScript web applications to automated testers. In Section 2.4, we distil these challenges into a set of seven criteria that an automated testing technique must satisfy in order to effectively test full-stack JavaScript web applications.

We identify the following main challenges:

Dynamic nature of JavaScript (Section 2.3.1) JavaScript is not only almost universally used to implement frontends of web applications, but it is also a popular implementation language for the backends of these applications. However, the highly dynamic nature of this language, including dynamic code loading, dynamic typing, reflection, higher-order functions etc., poses challenges to automated testers. Furthermore, the behaviour of an individual code snippet, such as a source code file or an individual function, may be affected by its composition with other snippets. For example, one code snippet might add extra properties to the prototype of an object defined in another snippet.

Event-driven code (Section 2.3.2) Event-driven code can be found throughout many layers of a full-stack JavaScript web application's technology stack, ranging from event-driven code in the frontend that handles user interaction to message handlers in the backend that handle communication coming from one of the other processes. Automated testers must not only be able to execute code that is hidden behind an event or message handler, but must also be capable of detecting bugs that *only* arise when a specific sequence of events is triggered.

Handling different multiplicities (Section 2.3.3) A full-stack JavaScript web application may spawn several processes belonging to the same layers, e.g., multiple web clients may be spawned, with all of them connected to the same server. Certain bugs only arise in specific compositions, which poses problems to automated testers that are not equipped to handle different multiplicities.

Process interplay (Section 2.3.4) When testing a full-stack JavaScript web application, an automated tester may explore each process of the application in isolation from each other while mocking inputs from other processes. However, this ignores the specifics of the interplay between these processes, which may result in a loss of precision for the tester.

2.3.1 Dynamic Nature of JavaScript

Websites almost universally rely on JavaScript, or one of its derivatives such as TypeScript or CoffeeScript, to render a web page interactive for users, to perform various computations on the page and possibly visualise the results thereof, or to communicate data to a backend [132]. Because of its prevalence, an automated tester for full-stack JavaScript web applications should therefore be capable of executing JavaScript code and detecting faults that arose during the execution.

However, JavaScript's highly dynamic nature, featuring e.g., dynamic typing, runtime code loading via `require`, object prototyping and dynamic code evaluation via the `eval` construct, render static analysis of this language difficult [9]. Rather than statically reasoning about the program's behaviour, automated testing techniques can directly observe the program's execution. Even so, these idiosyncrasies pose several problems for automated testers.

For example, JavaScript takes a permissive stance towards nonsensical operations, such as allowing arithmetic comparators to be used on string representations of numbers. Since actual program crashes are hence relatively rare, it is difficult for automated testers to gauge whether a seemingly erroneous operation was or was not intended by the developer.

As another example, JavaScript's dynamic typing system forces automated testers to generate multiple types per program input, as the type of the input may affect the behaviour of the program. This in turn increases the number of test generations required to explore the application.

As a last example, testing code snippets in isolation from each other is complicated by the fact that properties may be dynamically added to or removed from object instances. Behaviour of a code snippet may depend on which properties are present in an object used in the snippet, which forces automated testers to consider additional, often implicit, constraints on the object's layout while testing individual snippets.

2.3.2 Event-driven Code

As evidenced by the CALCULATOR application presented in Section 2.1.2, full-stack JavaScript web applications commonly rely on event-driven code to process various sources of user and system events. For example, to realise highly interactive web pages, developers often include JavaScript event handlers in the application frontend to capture and process user events. Developers may also rely on message handlers to process communication between the various layers of the application's technology stack. For example, the Socket.IO framework used by the CALCULATOR application allows for the creation of message handlers that are automatically invoked when the process receives a message of a certain type.

When a program is explored by an automated tester, the tester must account for these events in order to execute and explore the code hidden behind these event handlers. This means that the tester must be able to automatically generate appropriate events in order to invoke their corresponding event handlers and execute the code accessible from within the handler. As the order in which these events occur at run time is generally non-deterministic, the scheduling of these events may give rise to unforeseen data races.

Additionally, certain parts of the application may only be reachable by a following a specific sequence of events. For example, many functionalities of an application may only be accessible to a user that is logged in. In order to validate these features, the tester therefore has to generate events for e.g., loading the login screen, entering a user's credentials, and clicking the login button.

Furthermore, developers may opt to register or deregister event handlers dynamically, dependent on whether certain conditions are met. This dynamic registration and deregistration of event handlers can even take place inside other event handlers. In order to explore a registered handler, the tester would therefore first have to ensure that the necessary conditions for registering the handler have been met, only then to actually trigger the corresponding event.

In short, testers are forced to not only explore the application's *input space*, but also its *event space*, leading to an exponential increase in the number of test runs required to fully explore the application.

2.3.3 Handling Different Multiplicities

A full-stack JavaScript web application consists of various processes, each corresponding to a layer in the application's technology stack. However, several instances of a process corresponding to the same stack layer may be spawned. For example, users may be able to access the full-stack JavaScript web application via multiple web clients, all connected to the same backend, and exchange information with each other through this backend. As an example, consider a collaborative drawing application such as the one introduced in Section 1.1 where the drawing canvas is stored on a server. Multiple clients may connect to the server to view and draw on this shared canvas. In such an application, certain behaviour may only be triggered when a specific number of clients connects to the server. This may give rise to composition-specific program faults in the application, as certain bugs on the client or the server may only manifest themselves when a minimum number of clients have connected. Automated testers should therefore be capable of considering various multiplicities of the application, e.g., by spawning several instances of the application's constituent processes.

2.3.4 Process Interplay

The processes constituting a full-stack JavaScript web application communicate with each other and hence affect each other's behaviour. These processes may be tested in isolation from each other, e.g., by mocking communications from other processes when testing a particular process. However, such a practice may introduce imprecision, leading to false positive error reports.

As an example, consider the CALCULATOR application again. When testing the server code in isolation, while mocking Socket.IO inputs from the client, an automated tester might report the uncaught unknown-operator exception which can be thrown on line 23 in Listing 2.3. However, reporting this as a possible error in the CALCULATOR application would not be precise, since it ignores the fact that the client processes filters out all expressions involving an unrecognised operator (line 14 in Listing 2.1), rather than communicating these to the server. In other words, the interplay between the frontend and the backend prevents this error from occurring in the CALCULATOR application.

When testing this application, an automated tester would therefore have to either consider the interplay between the processes or resort to reporting errors that may not actually be triggered when the application is run by a user.

2.4 Criteria for an Automated Tester for Full-stack JavaScript Web Applications

In the previous sections, we introduced full-stack JavaScript web applications, presented various types of automated testers, and outlined the challenges faced by automated testers for full-stack JavaScript web applications. We conclude this chapter by distilling from these four challenges a total of seven criteria that automated testers for full-stack JavaScript web applications should satisfy in order to be capable of successfully testing these applications.

2.4.1 Overview of Testing Criteria

We identify seven criteria based on the challenges for automated testing of full-stack JavaScript web applications described in the previous section. As a motivated reasoning for why we identify exactly these criteria, we again use the example of the CALCULATOR application as a representative full-stack JavaScript web application, and we argue why each criterion is necessary to automatically test CALCULATOR.

Automated testers for full-stack JavaScript web applications should satisfy the following criteria:

Criterion 1: Capable of Testing Web Clients

An automated tester for full-stack JavaScript web applications must necessarily be able to test a web client (i.e., the frontend) process and a server (i.e., the backend) process of a full-stack JavaScript web application. Because of the unique challenges faced by a tester in this context, we divide this main criterion into two subcriteria.

Criterion 1.A: Capable of Testing Sequential JavaScript Code

As outlined in Section 2.3.1, the wide prevalence of JavaScript in the development of web clients of full-stack JavaScript web applications force automated testers to be capable of testing JavaScript code. That is, the tester should be able to automatically execute JavaScript code and detect and report erroneous behaviour, while taking into account the various idiosyncrasies of the language, e.g., dynamic code evaluation, a dynamic type system, higher-order functions etc. (cf. Section 2.3.1), that render testing more difficult.

The frontend and backend of CALCULATOR are implemented in JavaScript. In order to test both components, an automated tester has to respect the semantics of the language.

Criterion 1.B: Models Inputs from the DOM

We require that automated testers for full-stack JavaScript web applications consider the DOM as a source of program inputs. Concretely, this means that the tester should be capable of recognising how user or system inputs may originate from DOM elements, from the user's interaction with the web page, or from other sources of environment or system inputs, such as readings of external sensors or internal clocks.

Consider for example the web page of the CALCULATOR application depicted in Figure 2.2. User inputs in this application originate either from the buttons clicked on the page (cf. Criterion 2) or from the text input field. Criterion 1.B stipulates that any input through which the user, system, or environment may affect the behaviour of the application should be considered a program input. This includes the contents of input fields on web pages, selection of radio buttons or checkboxes, exact coordinates of a mouse click, external API calls that return non-deterministic values etc.

The frontend of CALCULATOR relies on a DOM featuring e.g., buttons which, when clicked, invoke the associated event handler. An automated tester for this frontend can only execute a handler by quering the DOM and simulating a click on the corresponding button.

Criterion 2: Capable of Testing Event-driven Code

Event-driven code is omnipresent in the development of full-stack JavaScript web applications, ranging from event handlers used for rendering web pages interactive to message handlers used for intercepting communication between the processes of the full-stack JavaScript web application. An automated tester for these applications should therefore be capable of executing event-driven code.

In the example of the CALCULATOR application, two types of handlers were defined: user event handlers that were invoked when a button on the web page was clicked, and message handlers from the Socket.IO framework which were executed when a Socket.IO message arrived at the client or server.

We decompose this main criterion into two subcriteria:

Criterion 2.A: Allows for Dynamic (De)registration of Event Handlers

In the case of an application's frontend, event handlers may be defined statically, in the HTML document describing the DOM of the web page, or dynamically via JavaScript. Dynamic registration of event handlers is more flexible, as it allows for registering or deregistering a handler when certain conditions are met. In order to explore *all* event handlers of an application, the tester should therefore take into account the dynamic registration and deregistration of event handlers. If registration or deregistration of an event handler is conditional, the tester should strive to satisfy the necessary conditions so that it can trigger the corresponding event and start exploring the event handler.

Consider lines 5 to 7 of Listing 2.1. In this snippet, several event handlers are registered dynamically by invoking the `addEventListener` method: clicking the corresponding button only has an effect after this method has been invoked.

Criterion 2.B: Explores Event Space

Events can generally be triggered in any arbitrary order. In some cases, this may lead to race conditions. For example, if event handler A writes to a variable that is read by event handler B, then the behaviour of the application likely depends on whether event A was scheduled before or after event B. It is therefore necessary for automated testers to fully explore the event space (cf. Section 2.3.2) of the application, by considering all appropriate orderings in which events can take place.

In the case of the CALCULATOR application, the frontend only sends a computation request to the backend if a specific sequence of events has been followed: i.e., creating the arithmetic expression by clicking number buttons and operator buttons, concluded by a click on the button labelled with the equality sign.

Criterion 3: Capable of Finding Composition-specific Faults

Section 2.3.3 discussed how some failures may only arise in certain compositions of the full-stack JavaScript web application. For example, consider a collaborative drawing application where multiple clients connect to a server in order to share a drawing canvas. The server may only expect a maximum of n clients to connect, and might fail when an $(n+1)$st client connects. In order to find all program faults in a full-stack JavaScript web application, an automated tester should therefore be capable of executing the same application with a different number of instances for client and server processes.

Criterion 4: Handles Process Interplay Precisely

The last main criterion concerns the interplay between all processes in a full-stack JavaScript web application. Rather than only testing the individual processes of the application in isolation from each other and mocking communication between them, we require the tester to be capable of verifying the interplay between the processes as well. This increases precision of the tester and prevents the tester from over-approximating the behaviour of other processes when testing one process in isolation. The usefulness of this increased precision was demonstrated in Section 2.3.4, which highlighted how testing the backend of the CALCULATOR application in isolation, while mocking messages from the frontend, would cause the tester to mistakenly report the uncaught unknown-operator exception on line 23 in Listing 2.3, even though the client filters out inputs leading to such a result.

We decompose this criterion into two subcriteria:

Criterion 4.A: Whole-Program Monitoring

In order to test the interplay between processes and consider how the actions of one process affect the behaviour of another, an automated tester should be capable of monitoring and driving the execution of all processes of the application, rather than just one single process.

Criterion 4.B: Observes Communication between Distributed Processes

To test the interplay between processes and reason about the impact of one process on the behaviour of another, the tester should be able to observe the communication between processes. As the communication directly represents the interplay between distributed processes, it is important for an automated tester to understand when these processes communicate and which data is sent from one party to another.

2.4.2 Summary of Testing Criteria

We identified four main criteria, three of which we decomposed into two subcriteria each, for a total of seven criteria. We repeat the complete list of criteria below:

1. Capable of testing web clients:
   - A. Capable of testing sequential JavaScript code
   - B. Models inputs from the DOM
2. Capable of testing event-driven code
   - A. Allows for dynamic (de)registration of event handlers

   B. Explores event space

3. Capable of finding composition-specific faults

4. Handles process interplay precisely

   A. Whole-program monitoring

   B. Observes communication between distributed processes

In the next chapter, we provide an overview of existing, state-of-the-art concolic testers and evaluate them on these testing criteria in order to study to what extent existing automated testers are capable of testing full-stack JavaScript web applications.

3 State of the Art in Concolic Testing

Chapter 2 defined testing of full-stack JavaScript web applications as lying on the intersection between testing of distributed systems, of event-driven applications, and of web clients and servers. In this chapter, we contextualise StackFul, the concolic tester for full-stack JavaScript web applications introduced in this dissertation, by giving an overview of past and ongoing research into concolic testing of these types of applications, and we sketch the wider problem and solution space in which StackFul is embedded.

We will discuss related work spanning the following domains:

Concolic testing of distributed systems (Section 3.1) Full-stack JavaScript web applications are distributed systems, where the client and server processes communicate with each other. It is therefore natural to consider the state of the art in concolic testing of distributed systems.

Concolic testing of event-driven applications (Section 3.2) As previously discussed in Chapter 2, both the client and the server side of a full-stack JavaScript web application make intensive use of event handlers for processing user and system events. Understanding how to perform concolic testing on event-driven applications efficiently is therefore essential for developing a concolic tester for full-stack JavaScript web applications.

Concolic testing of JavaScript applications (Section 3.3) Both the client and the server process in the full-stack JavaScript web applications targeted by StackFul primarily employ JavaScript, which poses a unique set of challenges to automated testing (cf. Section 2.4). StackFul must therefore draw inspiration from JavaScript-specific concolic testers.

Concolic testing of web servers (Section 3.4) Although the challenges that arise when performing concolic testing of web servers overlap with the challenges posed by testing distributed systems, testing the web server of an application introduces additional challenges. We therefore also cover state-of-the-art that is specific to concolic testing of server-side code.

Additionally, in Section 3.5 we discuss some generic optimisations to render concolic testing more effective. These optimisations are not specific to any type of application but can rather be applied to any concolic tester.

Naturally, the categories described in Sections 3.1 to 3.4 may overlap to some extent: a concolic tester for a web client may need to be capable of testing event-driven JavaScript code, and could hence be described in Section 3.2 or Section 3.3. When describing a state-of-the-art concolic tester in this chapter, we use our best judgement to decide which section most adequately describes the contributions provided by the tester. Nevertheless, in Section 3.6, we identify the concolic testers which are most suited to testing full-stack JavaScript web applications. We evaluate these testers on the list of criteria defined in Section 2.4 and summarise to what extent these testers are capable of testing full-stack JavaScript web applications, and in which aspects they are still lacking.

3.1 Concolic Testing of Distributed Systems

Regarding concolic testing of distributed systems, we consider actor-based applications and applications using the Message Passing Interface (MPI) [36]. Full-stack JavaScript web applications, distributed actor applications, and MPI applications have many commonalities. They consist of a set of processes where each process evaluates their own, unique sequence of branch conditions, resulting in per-process path constraints (cf. Section 2.2.3), and can send messages to and receive messages from other processes. Because the scheduling of message sends and receives is non-deterministic and prone to result in subtle data races, testers for distributed applications should consider various message schedules. This is similar to the non-deterministic event sequences which have to be controlled for in a concolic tester of full-stack JavaScript web applications.

Another important similarity between full-stack JavaScript web applications and these distributed systems arises from the payload of the messages sent between processes. The values extracted from within the payload of a received message are likely to affect the outcomes of the branch conditions evaluated by the receiving process. As these values encode symbolic expressions that originate from computations spanning one or more processes, it is important for the precision of the concolic tester that these symbolic expressions are included into the path constraints of the receiving process.

3.1.1 Actor-based Programs

Some concolic testers for actor-based languages have been developed in recent years. Vidal presents a concolic tester for a subset of the Erlang language, featuring message sending and receiving, pattern matching, and higher-order functions [130]. However, their description of the tester is limited. For example, there is no description of how different message schedules are explored. There is also no indication that the concrete values sent via messages to other actors correspond to some symbolic expression. It is possible that each value received by an actor is associated with an unconstrained symbolic input, and that all symbolic information on a value is therefore lost when it is sent to another actor.

TAP [75] is a symbolic execution engine for applications that use the Java API of the Akka framework[1], which enables stream-based and actor-based programming. Rather than performing traditional concolic testing on the actor application, TAP performs targeted *backwards symbolic execution.* Backwards Symbolic Execution. (BSE) was previously introduced by Ma *et al.* in testing of traditional, non-distributed applications [82]. In this setting, symbolic execution is targeted to a specific line or statement in the application. Backwards symbolic execution starts from the beginning of the function that contains the targeted line or statement and searches for a path constraint within the function that reaches the target. Upon finding the target, BSE iteratively jumps backwards in the call-chain, using a statically computed call-graph, concatenating the function-specific path constraints, until it reaches the start of the application with a satisfiable path constraint for the entire application.

TAP employs an actor-specific variant of BSE. Rather than operating on the call-graph, TAP first creates a message flow graph that captures the interactions between actors. It then uses this graph to perform backwards symbolic execution from a target line to an entry point of the actor system. TAP has to consider spawning new actors, which may lead to an exponential number of program paths being generated. To mitigate this exponential blow-up, TAP employs two optimisations related to finding the appropriate message type to be received in order to reach the target. This helps minimise the number of possible actors involved in an interaction.

Since backwards symbolic execution is a form of static symbolic execution and cannot be applied to concolic testing, TAP cannot be classified as a concolic tester.

[1]`https://akka.io/`

Symbolic execution for the Erlang language was also previously proposed by Earle [42], but this was also limited to static symbolic execution.

3.1.2 Message-passing Applications

Sen and Agha [114] presented an algorithm for concolic testing of message-passing applications, with the aim of covering reachable code and detecting deadlocks. Their algorithm is implemented in the jCUTE tester [115] for Java applications. Aside from exploring the value space for program inputs of the application, their technique must also account for non-determinism caused by both the scheduling of processes and the scheduling of message deliveries.

Sen and Agha use symbolic execution to generate data inputs that cover branch conditions that depend on these inputs, and use concrete execution to observe the order of execution events along the executed program path. This order corresponds to a *happens-before relation*, which defines causal connections between execution events, such as the sending of a message in one actor resulting in the message being received by another. Although many orderings of execution events are possible, this happens-before relation enables Sen and Agha to perform a partial order reduction to filter out equivalent orderings in an attempt to limit the number of schedules to explore.

Despite the similarities between the exploration of non-deterministic message schedules and event sequences, Sen and Agha's technique cannot easily be lifted to the exploration of event-driven code. In message passing applications, a message handler is invoked only after a corresponding message sent has been observed while executing the code. In event-driven applications, there is no explicit event loop which drives the instantiation of new events. The tester itself must rather construct and trigger event sequences based on which event handlers are available.

COMPI [73] is another concolic tester for SPMD (single program, multiple data) message-passing applications, implemented on top of the CREST tester [20] for C applications. It tackles two problems specific to MPI applications: solving for the *rank* of an application, and solving for the *number* of processes in the applications. An MPI rank is a unique identifier for a process of the application. This rank is a first-class value that may affect the process's execution. The total number of processes may also be queried by a process, which may influence its execution. By rendering the rank and number of processes as symbolic variables, they can be solved for, allowing for an increased code coverage.

COMPI performs symbolic execution on and generates inputs for only a single process, called the *focus*, of the system, but records branch coverage across all processes. The symbolic representations of the number of processes and the process rank may appear in the path constraint. COMPI can hence choose a different number of processes to be launched and a different process to serve as a focus before the start of each test run.

COMPI applies three optimisations for rendering concolic testing of an MPI application more scalable: it limits the size of generated inputs, it filters out redundant constraints generated inside loops, and it reduces performance overhead via *two-way instrumentation*. This last optimisation involves running the heavyweight instrumentation necessary for performing symbolic execution only on the focus, while having the non-focus processes run only the lightweight instrumentation that is necessary to record branch coverage.

The decision to perform symbolic execution on a single process only per test run renders COMPI capable of quickly covering a large part of an MPI application. However, this comes at the cost of precision, as COMPI cannot build a path constraint that spans multiple processes. Likewise, when a value is sent from one process to another, all symbolic information about the value is lost. This makes COMPI unsuited for precise testing of full-stack JavaScript web applications.

3.1.3 KleeNet

KleeNet [109, 110] is a dynamic symbolic execution engine for testing wireless sensor network applications, built on the KLEE symbolic execution engine [23]. KleeNet exhaustively explores an application by rendering environment inputs such as sensor data or packets' payloads as symbolic inputs. Furthermore, it injects non-deterministic failures, such as node reboots or dropped or duplicated messages, into the system to find errors in the program that occur due to node or network failures. Developers can furthermore check the correctness of the distributed system by formulating distributed assertions over the state of the entire system.

KleeNet employs node models and network models to express non-deterministic system failures [109]. The node model is used to represent failures of individual nodes such as node outages and node reboots. The network model describes the interaction between nodes.

Executing JavaScript Code

KleeNet is built on the KLEE symbolic execution engine and hence operates on LLVM bytecode. The ability of KleeNet to test JavaScript code hence depends on the possibility of compiling JavaScript to LLVM bytecode. Note, however, that KleeNet also requires models of the system's nodes and network to exhaustively explore the application's search space. This information may get lost when compiling high-level JavaScript source code to low-level LLVM bytecode.

Event Space Exploration

Although KleeNet has a notion of the application's event space, this space is defined only by the non-deterministic node and network failures that may occur in the program. KleeNet does not focus on event-driven applications where developers register an event handler or message handler that is invoked when the corresponding event or message is triggered (cf. Listing 2.1). This is an important distinction, as full-stack JavaScript web applications allow for dynamically registering or deregistering handlers both conditionally and within other handlers. In order to fully explore the search space of even a single handler in these applications, an automated tester must hence first ensure that the necessary conditions for registering and invoking the handler have been satisfied.

This requirement does not exist for non-deterministic failure events, such as those considered by KleeNet, as these failures may occur at any time. Nevertheless, within the scope of non-deterministic failure events, KleeNet is capable of exhaustively exploring the event space of an application. In order to model program paths that depend on these failures, KleeNet inserts nodes corresponding to the failures into the application's symbolic execution tree.

Client-Server Interaction

KleeNet is designed to test distributed systems in the form of wireless sensor network applications, and is capable of observing the interaction between processes. Users may configure the number and type of processes to be tested in the system, as the tool is not restricted to just one configuration of a client and a server process.

KleeNet builds a separate symbolic execution tree for each of these processes. However, in a distributed system where processes communicate with each other, a process `A` may affect the execution of a process `B` by sending `B` a message. The causal link between both processes must be reflected in their symbolic execution trees. For example, if process `A` conditionally sends a message to process `B`, the execution path of `A` should fork to reflect this decision. Likewise, one or more nodes in the symbolic execution tree of `B` may be duplicated to model the scenarios where the message was or was not received.

Because this quickly explodes the number of nodes in all symbolic execution trees, KleeNet provides three ad-hoc approaches for reducing the number of duplicated nodes across all symbolic execution trees [110]:

1. **Brute-force Copy-on-Branch** collects all nodes across the processes' symbolic execution trees in *distributed scenarios*, where a scenario is a set consisting of one node per process. Whenever a node in a distributed scenario is forked, for even a local branch, the remaining nodes in the scenario are forked as well. The number of nodes duplicated in even simple applications is hence extremely high.

2. **Delayed Copy-on-Write** broadens the concept of distributed scenarios to *distributed state*, which may include more than one node of a process, as long as the communication history of these nodes is consistent. Delayed Copy-on-Write allows for eliminating some redundant forking of nodes, but may still inadvertently fork the nodes of a bystander process `C` that is included in the same distributed state as the nodes of two communicating processes `A` and `B`.

3. **Super Distributed States** eliminates the redundant forking of these bystander nodes as well by allowing states to be in several distributed states and forking only target states, i.e., the states that receive a package from the sender.

3.2 Concolic Testing of Event-Driven Applications

We consider concolic testers for event-driven applications because of the central importance of event handlers in full-stack JavaScript web applications. A large part of the code of both the client and the server is only accessible by triggering specific events. Additionally, a tester must often not only generate an appropriate *individual* event, but it must also generate a specific *sequence* of events: i.e., it must, efficiently, explore the event space of the application.

Since event-driven code is ubiquitous in a large variety of different applications, we consider concolic testers for both event-driven web clients as well as event-driven mobile applications.

3.2.1 Cosette & JaVerT.Click

Santos *et al.* developed *Cosette* [108], a symbolic execution engine for plain JavaScript applications. *JaVerT.Click* [106] builds upon this work by adding support for event handlers and modelling the DOM, enabling symbolic execution of web clients.

Executing JavaScript Code

Cosette first compiles JavaScript to JSIL bytecode via JS-2-JSIL [107]. Santos *et al.* offer semantics for the concrete and symbolic execution of this bytecode, as well as a reference implementation for executing these semantics. Although Cosette by itself cannot perform concolic testing, the combination of the concrete and symbolic semantics opens up the possibility of building a concolic tester by extending the reference implementation. Santos *et al.* prove the bounded soundness of their symbolic semantics. To validate their implementation, they identified the relevant test cases of the official Test262 suite[2], and pass 100% of these selected tests.

As it is, Cosette can be used to verify developer-specified assertions in JavaScript programs by symbolically executing the application. For each assertion that is

[2]`https://github.com/tc39/test262`

executed by the engine, Cosette attempts to generate a concrete model of inputs that would cause the assertion to fail. Loops and recursive function applications are unfolded up to a certain bound to ensure that symbolic execution terminates.

Cosette supports symbolic strings, numbers, booleans, and heap locations. JavaScript libraries are compiled to JSIL as well, but are not separately modelled. Calls to library functions are therefore supported in a precise manner by Cosette, but they may result in a loss of performance.

Event Space Exploration

Cosette does not support triggering events and executing a sequence of event handlers. It is therefore unable to test web clients or any other event-driven applications. However, its extension, JaVerT.Click [106], provides concrete and symbolic semantics for a generic event model, called the Core Event Semantics. (CES).

The CES is parametrised with semantics for an underlying language model and by itself does not completely describe how an event-driven application can be executed. The semantics instead describes registration and deregistration of event handlers as well as synchronous and asynchronous dispatching of these handlers, while relying on the underlying language semantics to recognise when the appropriate event handler operation must be invoked. The CES assumes a synchronous event model with atomic event handlers. This is nevertheless sufficiently expressive to model Dom UI Events [131], JavaScript Promises [43] and JavaScript's async/await API [43].

Sampaio *et al.* [106] created a reference implementation in JaVerT.Click for these three instantiations, which remains faithful to the standard in which they are defined. This implementation also includes a complete model of the DOM Core Level 1 [28] to enable reasoning about web clients that interact with their DOM. Combining this infrastructure with the existing Cosette engine results in *Javert.Click*, which is capable of static symbolic execution of JavaScript applications that employ any of the three aforementioned event-driven APIs.

Like Cosette, JaVerT.Click by itself cannot be used to perform concolic testing. However, since Sampaio *et al.* define both concrete and symbolic semantics for the three instantiations of CES, extending JaVerT.Click to also perform concolic testing should be feasible. On the other hand, neither the CES itself nor its reference implementation in the form of JaVerT.Click describes *how* event-driven code should be executed. In order to efficiently test event-driven code while minimising the number of test runs required by the tester, it is necessary to generate *appropriate* event sequences. Without a means to generate these sequences efficiently, testing highly dynamic web applications, such as most full-stack JavaScript web applications, remains infeasible.

Client-server Interaction

Both Cosette and JaVerT.Click test only single, individual JavaScript processes, rather than a distributed application consisting of multiple communicating JavaScript processes. Several changes would therefore have to be made to enable either symbolic execution engine to perform testing of a full-stack JavaScript web application.

For one, although JaVerT.Click is capable of symbolically executing plain JavaScript applications, such as Node.js server processes, it does not yet support modelling communication received or sent by the server. JaVerT.Click would therefore have to instantiate the CES to support e.g., Socket.IO-style message handlers. Since the CES has been proven to be sufficiently powerful to model a range of different event models, it should be feasible to extend JaVerT.Click with the ability to do so.

The inability of both Cosette and JaVerT.Click to test two processes simultaneously constitutes a larger obstacle, however. None of the proposed semantics includes a mechanism for performing symbolic execution across more than one process. It is unclear how either engine could be extended to support testing distributed applications. We therefore conclude that neither Cosette nor JaVerT.Click is suitable for concolic testing of full-stack JavaScript web applications.

3.2.2 SymJS

SymJS [72] performs automated testing of dynamic web clients and explores both the client's value space and event space.

Executing JavaScript Code

SymJS performs concrete and symbolic execution of both standalone JavaScript applications as well as client-side web scripts. JavaScript programs are first compiled into Rhino bytecode[3] and then executed via a virtual machine. Li *et al.* [72] present semantics for the symbolic execution of this bytecode. The semantics are implemented in the virtual machine, along with a symbolic DOM model, so that it may perform concrete and symbolic execution of the application. The engine has also been extended with the ability to perform dynamic taint analysis.

SymJS employs the PASS solver [71] for solving constraints involving integers, floating point numbers, strings, and regular expressions. Although string constraints have traditionally been challenging to solve, the use of PASS makes SymJS capable of supporting the majority of JavaScript's string operations.

SymJS uses HTMLUnit's[4] DOM and browser API model for modelling concrete interactions with these respective APIs and extends it with support for symbolic reasoning. This makes it possible to include some UI elements, e.g., form fields, as input values.

Furthermore, SymJS provides a symbolic model of some commonly used JavaScript libraries for computations and data structures, such as `Number`, `Array`, and `Math`.

Event Space Exploration

SymJS explores the event space in addition to an application's value space. To this end, SymJS offers several strategies for exploring the event space. The simplest of these strategies amounts to generating and executing random sequences of events. The other strategies generate new event sequences from existing ones by concatenating individual events to the sequence. The strategies prioritise the exploration of event sequences that maximise the number of conflicting write-read dependencies on program variables that are shared between the events in the sequence. Li *et al.* present three different ways for measuring the dependency information:

[3]`http://mozilla.github.io/rhino/`
[4]`https://htmlunit.sourceforge.io/`

1. **Named WR** simply counts the number of variables that are written to by at least one event handler in the entire event sequence, and read from by the last event handler.

2. **Valued WR** is similar to Named WR, but only considers a write-read dependency on a variable v if the value written to v is different from any other value ever read from v.

3. **Taint Named WR** filters out redundant write-read conflicts by tainting variables to determine whether the variables are relevant to unvisited branches in the last event handler.

Client-Server Interaction

SymJS is capable of performing concolic testing on standalone JavaScript applications, in addition to client-side web scripts. However, SymJS only targets individual applications and does not tackle a distributed configuration of multiple JavaScript processes.

Although SymJS would therefore be capable of testing both the Node.js server side as well as the JavaScript client side, it cannot test both processes at the same time, nor monitor the interaction between the two. We therefore conclude that although the search strategies proposed by SymJS are useful to efficiently test a client's event space, SymJS itself lacks the ability to perform inter-process testing of a full-stack JavaScript web application.

3.2.3 Mobile Applications

Like web clients, many mobile applications are heavily driven by user interactions: e.g., pressing a button or performing an action may result in the mobile application loading a new context where a new code snippet will be executed. These mobile applications are equally challenging to test as event-driven applications.

However, mobile applications do not form a distributed system on their own. Concolic testers for mobile applications therefore do not have to address all challenges posed by a distributed system. The testers described in this section are therefore not suited for testing full-stack JavaScript web applications.

Contest [7] is a concolic tester for Android applications that considers both how to generate single events as well as how to construct entire event sequences. The former goal is complicated by the fact that the GUI of an Android app organises its graphical elements in the form of a DOM-like tree structure called the *view hierarchy*, which may consist of both standard and app-specific *widgets*. Existing approaches for interacting with these widgets are either model-based or use capture-replay. Model-based approaches require developers to provide a model describing each widget and are hence difficult to use. Capture-replay approaches may not distinguish between a user interaction targeting a widget in the view hierarchy or any of its children, even though the app itself distinguishes between both interactions. Contest therefore opts to instrument the Android SDK so that it can observe how user events, such as a tap on a target location, are handled by the system. As the SDK compares the tap's target coordinate with the location of the widgets in the view hierarchy to determine the innermost widget to dispatch the tap event on, Contest constructs path constraints that characterise how an individual event can be generated for each widget in the view hierarchy.

To achieve its second goal of constructing event sequences that efficiently cover the entire application, Contest prunes redundant event sequences by using subsumption checking. Contest states that an event sequence ending with event e is subsumed by the same sequence without this final e if the handler for e does not write to any variables used by other event handlers in the sequence. An event sequence that is subsumed by another sequence that has already been explored does not need to be tested again.

Collider [60] performs targeted concolic testing of Android applications. Given a target statement in the application, Collider attempts to build an event sequence that will reach this statement. To this end, Collider employs a two-phase technique. In the first phase, Collider uses concolic testing to build symbolic summaries of all individual events in the application. In the second phase, Collider uses a form of backwards symbolic execution to construct an event sequence that will result in execution reaching this target. Specifically, Collider combines the symbolic event summaries from the first phase with a UI model that defines event transitions and specifies the read-write dependencies between any two event handlers of the application. Collider can then construct an appropriate event sequence by starting from the target and chaining together individual paths in each summary until it reaches the initial state of the application. Once a feasible sequence has been generated, Collider executes the sequence concretely to verify that the sequence indeed reaches the target.

3.3 Concolic Testing of JavaScript Applications

JavaScript employs many language features that make it difficult to test (cf. Section 2.3.1), such as dynamic typing, object prototyping, and dynamic code generation [9]. Automated testers for JavaScript furthermore have to contend with JavaScript's permissive handling of what would be considered erroneous behaviour in other languages. For example, when JavaScript finds an unexpected type, it attempts to implicitly convert the value to another type, rather than throwing a type error. We therefore consider concolic testers for this language in detail. There have been several surveys on automated testers and dynamic analyses for JavaScript [9, 126].

3.3.1 Jalangi

Jalangi [116] is an instrumentation framework for implementing dynamic analyses in JavaScript. Jalangi operates in two phases: in the *record* phase, an execution of user-specified parts of the application (e.g., excluding third-party libraries) is recorded. In the subsequent *replay* phase, the dynamic analysis is performed by running a *shadow execution* on the recorded program trace. As a demonstration of the expressiveness of Jalangi, Sen *et al.* implemented several dynamic analyses, including concolic testing.

The shadow execution operates on *shadow values*: wrappers for regular JavaScript values which carry extra, analysis-dependent information to enable the analysis to collect and compute the required information. In the case of the concolic testing implemented in Jalangi, this extra information comes in the form of symbolic expressions describing the computations which resulted in these concrete values.

Since Jalangi's concolic tester was developed only to demonstrate the usability of the Jalangi instrumentation platform, the tester is less feature-complete than others described in this chapter. For example, the tester cannot explore the event space of the web application. It is limited to executing only the program trace that was observed in the recording phase. Any event handler that was not invoked during this phase will hence be left unexplored.

Jalangi furthermore only observes the execution of a single, individual JavaScript process. It cannot perform concolic testing over an entire distributed system, as is required to perform concolic testing of a full-stack JavaScript web application.

3.3.2 Kudzu

Saxena *et al.* developed Kudzu [111], to find client-side code injection vulnerabilities in web applications. It uses concolic testing to explore the client while looking for program paths on which improperly sanitised inputs flow into critical sinks such as `document.write`. To this end, Kudzu first records a concrete execution of the client and afterwards re-executes this recording symbolically, in order to explore more program paths.

While exploring a program path, Kudzu identifies several sources of inputs: user data (originating from DOM elements such as input fields), HTTP channels (through which the client communicates with a web server), and locations where different windows or frames of the page communicate with each other. Although the client may attempt to sanitise this data before having it flow into a sink, Kudzu still has to verify whether this sanitisation was complete. When Kudzu observes an input string flowing into a sink, it therefore matches the (potentially transformed) input string against a sink-dependent regular expression which describes possible formats a successful attack string may take. If Kudzu's solver determines that a concrete string could be generated which, after passing through the client's input transformation can still be matched against the regular expression, Kudzu replays the execution, has the solver-crafted input string flow into the sink and reports the vulnerability if the attack is successful.

Kudzu was built on FLAX [112], which also performs taint tracking of inputs. However, FLAX was restricted to taint analysis in manually-written test harnesses. The main contribution of Kudzu is therefore the automatic exploration of the client, making it possible to find code injection vulnerabilities without requiring the developer to intervene.

Executing JavaScript Code

Kudzu is implemented as an extension to the WebKit browser's JavaScript execution engine. At first, it only records a concrete execution of the web application. This recording is later re-executed symbolically, in order to explore more program paths. Kudzu supports symbolic booleans, integers, strings, and regular expressions. Kudzu employs a custom constraint language to describe the most commonly used JavaScript operations on these values. As at the time of Kudzu's development no off-the-shelf solver was sufficiently powerful to tackle all of these constraints, especially those describing string operations, Saxena *et al.* created their own solver.

Like Cosette and JaVerT.Click, Kudzu does not model third-party JavaScript libraries. Performance slowdowns may therefore be expected when using Kudzu to test JavaScript applications that employ such libraries.

Event Space Exploration

Since many clients include event-driven code, Kudzu recognises the need to traverse both the client's event space in addition to its value space. Kudzu generates random orderings of currently enabled events. Although this makes it trivial to take into account dynamic registration of event handlers, the exploration of the event space becomes suboptimal. For instance, Kudzu does not actively attempt to invoke event handlers that have not yet been activated. Some event handlers may hence remain unexplored for a long time.

Client-Server Interaction

Kudzu only performs concolic testing on the client side of the web application. It is hence not capable of performing testing of a distributed system, such as a full-stack JavaScript web application.

Kudzu considers HTTP channels through which the client sends data to or receives data from the server, as respectively sources and sinks of the JavaScript application. Kudzu does not monitor how data is sent across the client-server boundary, so it does not have a complete picture of which data will be sent in the application in practice. As Saxena *et al.* acknowledge, false positive alerts may be raised for client-side code injection vulnerabilities, since the server may perform its own sanitisation of its data before sending it to the client. We therefore conclude that Kudzu is not sufficiently precise to perform concolic testing of full-stack JavaScript web applications.

3.3.3 ArtForm

ArtForm [123] is a tool for performing dynamic analysis on websites where user interaction is heavily reliant on the user entering data via text fields on the website.

ArtForm includes three exploration modes for analysing such websites. Each of these modes represents a different way in which data is entered into a text field of the website. In *basic mode*, the user manually fills in the field, while ArtForm generates a trace and coverage report. ArtForm employs symbolic execution to detail how the user inputs affected the control flow and data flow of the website.

In *concolic mode*, ArtForm explores the website fully automatically by recording the path constraint of the execution, negating and solving the path constraint to obtain appropriate string values for the text fields, and re-executing the website.

The *advise mode* forms a middle-ground between the fully manual basic mode and the fully automatic concolic mode. Users manually enter data into text fields but can ask ArtForm for an appropriate value which would force execution along a previously unexplored program path.

ArtForm is built on Artemis [11], and hence re-uses many of its features for generating event sequences. This enables ArtForm to fully explore websites where event handlers are (dynamically) registered for user interactions performed on text fields.

However, ArtForm suffers from the same deficiencies as e.g., SymJS and Jalangi in that ArtForm only considers testing the client side of the web application. Since ArtForm only provides an overview of how execution flows through the client, it cannot provide a global overview of the entire full-stack JavaScript web application.

3.3.4 ExpoSE

ExpoSE [80, 81] is a concolic tester for Node.js with a complete model of ECMAScript 2015's regular expression language. This language is strictly more powerful than a regular expression language that can be encoded by finite-state automata, as ECMA regular expressions also allow for defining capture groups and backreferences.

ExpoSE mostly focuses on supporting JavaScript regular expressions to the fullest extent possible, and does not seem to offer support for any other aspect of the language not already covered by one of the JavaScript concolic testers mentioned in this section.

3.4 Concolic Testing of Web Servers

Since full-stack JavaScript web application testing covers the client and the server, we also consider concolic testers for web servers. Traditionally, testers for web servers do not focus specifically on testing servers in a full-stack setting. Nevertheless, they face challenges that are also relevant in the context of full-stack JavaScript web applications, such as how to explore message handlers that are invoked upon receiving a message from the client.

This section discusses CRAXWeb [120] and Apollo [10], which are both specifically designed to test web servers. However, there are many other testers that, although not specifically designed to test web servers, have nevertheless been applied to this setting. Examples include SymbexNet [122], which is used to test network protocol implementations, and EXE [22].

3.4.1 CRAXWeb

CRAXWeb [120] finds security violations, in the form of cross-side scripting attacks and SQL injections, in the server side of web applications.

Although CRAXWeb does not explore the client side of these applications, it uses the Acutenix[5] web crawler to find client-side code points where the client communicates with the server by sending an HTTP request.

After finding these request sites, CRAXWeb tests the server by automatically generating the corresponding requests, with symbolic data embedded inside them. To mitigate the path explosion problem, where the number of paths to be explored by the tester is exponential in function of the number of branching points encountered, CRAXWeb by default does not apply full-scale concolic testing on the server, but employs *single-path concolic mode*.

[5]`https://www.acunetix.com/`

This mode follows execution along one particular path only, which is determined by the message's concrete payload: a string filled with garbage. Symbolic constraints encountered during the execution of the path are collected, but are generally not used to explore alternative paths in later test runs. Rather, when an endpoint is reached where the server is observed to either send an HTTP response to the client or perform a database query, the symbolic constraints are passed to a solver which attempts to formulate an *attack string*. Such a string represents respectively a cross-site scripting attack performed on the client, or a SQL injection attack on the database. If an attack string can be formulated by the solver, it is possible for an attacker to send a payload to the server containing appropriate values that bypasses all of the server's checks and results in an attack.

Executing JavaScript Code

CRAXWeb is built on the S^2E [31] symbolic execution framework, which operates on native machine code and is capable of modelling even parts of the host's operating system environment. The ability of CRAXWeb to test JavaScript code therefore hinges on whether the code may be compiled to a format that is executable by S^2E.

Event Space Exploration

As mentioned before, the Acutenix web crawler is used to crawl the client-side code to look for HTTP requests that are sent to the server. CRAXWeb itself does not explore the client at all. There is hence also no need to generate appropriate event sequences to test the client.

Client-Server Interaction

Interaction is modelled solely through the HTTP requests found by the web crawler. Any HTTP request later mocked by CRAXWeb therefore represents a request that may actually be sent by the client. However, CRAXWeb does not monitor the interaction between the client and the server. Client-side constraints that would be observed along the path in which a request is found, can therefore not be joined with the server-side constraints that would be observed upon receiving a request. False positive attacks may therefore be reported, since the client could filter out inappropriate strings before passing them to the server. We therefore conclude that CRAXWeb is not capable of precisely testing a full-stack JavaScript web application.

3.4.2 Apollo

Apollo [10] uses concolic testing to test PHP server-side code with the aim of detecting malformed HTML pages being generated, as well as other, PHP-specific run-time errors.

Apollo is implemented as a PHP engine extension and uses shadow execution to associate concrete values with symbolic input variables. Symbolic inputs to the server-side code may originate from the following sources:

- Values retrieved from the web application's database.
- Parameters passed from the client to the server via HTTP GET and POST requests.

Unlike other symbolic execution engines, Apollo does not collect sophisticated constraints on the values appearing in branch conditions.

Rather, each concrete value is either associated with a symbolic variable *as is* or it is not. For example, if a branch condition relies on a substring of a symbolic variable, rather than the entire string, then Apollo cannot track from which symbolic input the substring was computed. This makes Apollo incomplete, as it cannot force execution along one of both branches of such a branch condition. This problem is partly mitigated by having Apollo mine the application's code base for literal values. When an unconstrained value is instantiated, it receives either a random value or a literal value appearing somewhere in the code base.

After completing testing and producing a bug report, Apollo attempts to minimise the path constraints that led to each reported fault. This may help developers understand how the fault can be triggered. This minimisation is achieved by first taking the intersection of all path constraints that expose a particular fault, and then iteratively removing individual constraints from this intersection and re-running the application until the fault is no longer triggered.

Executing JavaScript Code

Apollo only targets server-side PHP code. It cannot execute or monitor the execution of any JavaScript code, including the JavaScript code contained inside the HTML snippets generated by the server. Since these JavaScript snippets might pass values to the server, certain interactions between the client and the server which affect the execution of the server-side code go unnoticed.

Event Space Exploration

Support for modelling user events in the web application is very rudimentary. As the client side of the web application is not modelled at all, Apollo cannot trigger client-side user events or system events. Message handlers on the server side of the application are also not automatically invoked.

Artzi *et al.* discuss the possibility of manipulating Apollo into considering a limited form of user interaction. Users of Apollo can manually change locations in the PHP code where web pages containing buttons linking to other web pages are generated so that, instead of generating the HTML page, a switch statement on a symbolic input is executed. Every branch in the switch statement should result in the loading of one web page which was linked to by a button in the original HTML page.

Although this effort-intensive method allows Apollo to explore a larger fraction of the web application than it could otherwise, it is still a poor substitute for an actual exploration of the application's event space. Furthermore, it remains impossible to trigger user events that do not result in a new web page being loaded but that have some other effect, such as a change in the state of the client.

Client-Server Interaction

Apollo only targets individual server-side processes and as such is incapable of testing an entire distributed system. However, its ability to represent HTTP communication from the client as symbolic input makes it at least able to over-approximate messages and values sent by the client, which in turn allows Apollo to explore behaviours of the server that rely on certain messages or values being sent by the client. Apollo still completely neglects the client side of the web application, however, and is therefore incapable of finding bugs on the client side of the application, leaving Apollo unsuited to perform concolic testing of full-stack JavaScript web applications.

3.5 Optimisations for Concolic Testing

Several techniques have been proposed to render concolic testing more effective, that is, to have concolic testing achieve a greater code or program fault coverage in equal or fewer test runs. In this section, we list generic optimisation techniques that could be applied to any concolic tester, regardless of the type of application that is targeted by the tester. These techniques are therefore orthogonal to any language-specific or application-specific algorithms that may be employed by a concolic tester described in the previous sections. We categorise these generic optimisation techniques in function of *how* they attempt to improve the concolic tester:

Heuristical techniques (Section 3.5.1) These techniques employ heuristics to select new, unexplored program paths. These heuristics assign a greater priority to certain program paths which exhibit particular characteristics. For example, new heuristics may prioritise program paths that lead to newly added or updated parts of the code.

Path explosion mitigation techniques (Section 3.5.2) These techniques reduce the search space of the program by mitigating the path explosion problem. This can be achieved by e.g., merging similar program paths together or reusing symbolic summaries for parts of the application, rather than re-executing these parts.

Hybrid techniques (Section 3.5.3) Hybrid techniques combine concolic testing with other forms of program verification, e.g., static analyses or other types of fuzzing. These techniques attempt to achieve a mix where the benefits of one type of verification (e.g., fast generation of random inputs) are combined with the benefits of concolic testing (e.g., complete exploration of program paths).

Some approaches may use elements from more than one of these categories, e.g., an optimisation that describes a novel heuristic may depend on a static analysis to collect information for the heuristic, and could hence be placed in more than one category. We use our best judgement to decide how to categorise a technique.

3.5.1 Heuristical Techniques

Heuristical techniques generally direct the concolic tester to prioritise testing of certain inputs. The heuristic may deem that these inputs will lead the tester to more quickly reach parts of the program that are considered important, such as as-yet uncovered code, newly introduced parts of the code base or code modules that are known to be error-prone. Alternatively, the developer may decide to target a particular statement and the heuristic may have to consider how to reach this target in the least number of test test runs.

Several such search strategy heuristics have been developed in recent years [78]. We distinguish between *target-driven heuristics*, which guide the tester towards execution of a certain location in the code, and *coverage-driven heuristics* which direct the tester to maximise code coverage without specifying a particular target.

Target-driven Heuristics

Burnim and Sen [20] describe a heuristic which uses the control flow graph of the program to direct testing towards unspecified targets in the code that have not yet been uncovered. This heuristic uses a distance metric, based on the number of branch conditions that must be traversed, to compute the minimal distance between a part of the application that has not yet been covered and the nearest covered branch condition. This metric is then used to inform the concolic tester about which branch conditions must be negated in order to explore the closest uncovered part of the application. Similar techniques have been proposed that use an inter-procedural control flow graph to direct a concolic tester along the shortest path, according to some distance metric, to some target statement [82, 103]. Finex [137], implemented in PEX [128], likewise computes a fitness value that describes how close an already explored program path is to a particular statement, in order to guide the tester towards reaching that statement.

Coverage-driven Heuristics

SAGE [51] uses a heuristic that keeps track of the incremental code coverage obtained by a set of inputs and prioritises expanding on those inputs. Car-Fast [98] attempts to maximise coverage by exploring the uncovered branch on which the highest number of statements are control-dependent. Seo and Kim describe context-guided search, in which the tester considers the *context* of a branch: the branch conditions immediately preceding which had to be traversed in order to reach the first branch [118]. Their technique then prioritises exploring paths that lead to different contexts.

ParaDySE [26] considers 40 boolean features of a branch (e.g., whether the branch is located inside a loop, whether it has been explored in the last 10 test runs, and whether it appears in the main function of the program). These features are given a weight so that branches can be scored and the tester can select the branch with the highest score. Weights are specifically tailored to the target program and are automatically computed by an optimisation strategy which maximises branch coverage. This work was later expanded upon so that the tester can learn new heuristics and switch between existing ones while the program is being tested [25].

3.5.2 Path Explosion Mitigation Techniques

In general, the number of paths that should be explored by the tester is exponential in function of the number of branching points encountered during the program's execution. This problem is referred to as the *path explosion problem.* Path explosion mitigation techniques reduce the number of paths that are generated or stored in the symbolic execution tree. This can be achieved by reusing results produced during symbolic execution of parts of the application (i.e., *compositional symbolic execution*), or by explicitly merging paths in the symbolic execution tree together (i.e., *state merging*) [140].

Compositional Symbolic Execution

Godefroid presented compositional symbolic execution for producing symbolic summaries of individual components of a program [49, 6]. In the first application of this approach, these components corresponded to function definitions. With compositional symbolic execution of function, the tester prioritises fully exploring each function that is called in the application, rather than exploring any particular part of the code base. While exploring a function, the tester constructs a symbolic summary that describes the behaviour of the function. This function summary consists of a disjunction of the individual paths through the function, mapping specific pre-conditions on the function's inputs to the corresponding post-conditions on the function's outputs. When the function is later called again in the application, the tester can reuse this summary rather than performing symbolic execution on the function anew, which would result in more branching of program paths. Similar symbolic summaries can also be created for loops, with pre-conditions and post-conditions on the variables read from and written to inside the loop [50, 125, 138, 139].

Lin *et al.* propose to go even further than summarising functions or loops, by constructing summaries at the level of basic blocks [77]. Symbolic summaries at such a fine-grained level of control would allow for more reuse of these summaries, and prevent a large amount of calls to the SMT solver, which would be especially beneficial if some of the paths in a summary are actually infeasible.

Compositional symbolic execution can be combined with forms of target-driven testing (possibly using heuristics, cf. Section 3.5.1) to efficiently find feasible program paths leading to a particular target statement [95, 6, 102]. Some target-driven compositional testers use backwards symbolic execution. These testers usually operate in two phases: in the first phase, the tester constructs symbolic summaries, while in the second phase, the tester performs symbolic execution backwards on the program, starting from the target statement and continuing until the tester reaches an entry point into the program (cf. the TAP tester, Section 3.1.1) [96, 75].

Although all of the previously mentioned work attempt to infer these summaries automatically, symbolic specifications could also be defined manually [8].

State Merging

State merging is another method to limit path explosion [140]. State merging will be the main focus of Chapter 6, where we will discuss in detail how this technique can be applied to concolic testing of both sequential and event-driven code. In short, state merging addresses the path explosion problem by merging different *program states* which are sufficiently "similar", according to some similarity metric, together. A program state can be broadly thought of as being a node in the symbolic execution tree (cf. Section 2.2.3). In effect, this transforms the symbolic execution tree into a directed acyclic graph.

Program states are created alongside paths and represent a snapshot of the current point in the symbolic execution. Among other components (cf. Section 6.1), states include path constraints. An exponential increase in the number of paths therefore corresponds to a likewise exponential increase in the number of states and vice versa. Reducing the number of states by merging similar states together hence also mitigates the path explosion problem.

A downside to this approach is that the merged paths become harder to reason about. The symbolic execution engine must remain capable of precisely modelling how conditions on program inputs lead expressions and variables to evaluate to particular symbolic values. Expressions whose value depends on the path that was followed to reach the expression are usually represented as *if-then-else* (ITE) expressions. The ITE expression $ITE(cond, X, Y)$ represents an expression that evaluates to the symbolic value X if $cond$ is true, and to Y otherwise. However, these ITE expressions increase the burden on the queried SMT solver, and, when applied naively, may even lead to an overall performance loss [66].

Another problem is the fact that state merging may interfere with other optimisation techniques applied by the tester. State merging works best when similar states are merged as soon as possible, as this prevents the highest number of duplicate descendant states from being created. Ideally, if a state similar to one the tester has just explored is available, the tester would select that similar state so that it can be merged if found similar enough. However, the tester may employ a search strategy which prioritises exploration of a completely different state, for example because that state would lead the tester closer to a particular target. The tester therefore has to choose which optimisation technique to follow.

Kuznetsov *et al.* demonstrated two techniques for addressing these problems [66]:

- *Query count estimation* mitigates the burden placed on the SMT solver by taking into account the number of variables with differing values between two states when looking for similar states. Query count estimation attempts to minimise the number of ITE expressions that are created, as these expressions are computationally intensive to solve for the SMT solver. It does so by only merging states whose same variables with different values are expected to appear infrequently in future solver queries.

- *Dynamic state merging* does not interfere with existing search strategies but identifies opportune moments for merging dynamically. The tester's search strategy generally retains the privilege of selecting which state to explore next. However, if the dynamic state merging algorithm detects that a state scheduled for exploration at some future point will be sufficiently similar to an already explored state, the state merging algorithm overrides the decision of the search strategy and schedules the first state for immediate exploration.

MergePoint [12] alternates between static symbolic execution and concolic testing to take advantage of both approaches. It starts testing the program by performing concolic testing. When a branch condition is encountered, rather than forking the execution, MergePoint may switch to using static symbolic execution, since state merging can be applied more easily in static symbolic execution. While in static mode, MergePoint merges states together until performing static symbolic execution becomes too difficult, for example because of a system call or an indirect jump. At that point, MergePoint reverts to concolic testing and continues testing the program.

Jaffar *et al.* [59] use *interpolation* to prune program paths by terminating the exploration of certain paths early, before they can spawn new paths. With interpolation, the symbolic execution engine creates interpolants, which describe conditions necessary to trigger a particular bug. If during exploration of a path the engine can infer that the interpolant is not satisfied, it terminates exploration of the path.

As mentioned previously, using ITE expressions to represent merged symbolic values allows the tester to precisely model which values an expression evaluates to given a set of conditions, but this also increases the burden for the SMT solver. Rather than relying on ITE expressions, Scheurer *et al.* take inspiration from abstract interpretation to define a general lattice model [113]. When merging two values together, they perform a join operation on the values in the lattice and represent the merged value through this joined value. Although this loses precision, Scheurer *et al.* deem this an acceptable loss since queries to the SMT solver become easier to process.

MultiSE [117] takes the idea of merging program states even further, by using one shared, incrementally constructed representation for all execution states that have been explored by the tester. In this shared state, all variables, including the program counter, are described using *value summaries*. A value summary is a mapping of a set of conditions to the value that the variable will take if these conditions are true. In effect, MultiSE corresponds to merging program states at every assignment, rather than only when two states happen to collude.

3.5.3 Hybrid Techniques

Concolic testing can be combined with other forms of program verification to either traverse the search space of the application more quickly, or to reduce the size of the search space altogether. We distinguish between hybrid techniques where the tester is combined with a dynamic form of program verification, such as some different form of automated testing, and static techniques, where the tester employs a static analysis.

Dynamic Program Verification

Section 2.2 listed several types of automated testing techniques, including random input fuzzing and search-based techniques. As mentioned previously, a main advantage of concolic testing is its ability to systematically test all *feasible* paths through an application, whereas its disadvantage lies in the fact that it will take a long time to do so. Other forms of automated testing generally take the inverse approach, by prioritising the exploration of a large part of the program as quickly as possible, without having any guarantees that all reachable parts of the program have been covered. Both approaches can be combined into a hybrid approach in an attempt to combine the benefits of both techniques.

Majumdar and Sen propose to interleave concolic and random testing [83]. Their hybrid tester starts by generating random inputs for the program, but switches to concolic testing once the random tester is unable to cover new unexplored parts of the code. When the concolic tester reaches a previously uncovered part, the tester resumes random testing.

Wang *et al.* propose a strategy for alternating between concolic and random testing [135]. This strategy weighs the cost that a concolic tester incurs by constraint solving with the probability that a path will be explored by a random tester. This hybrid approach therefore uses random testing to cover paths that are likely to be explored with random inputs, and uses concolic testing to cover paths that are hard to reach.

Driller [124] distinguishes between *general* inputs, which have a wide range of valid values, and *specific* inputs, which are limited in the values they may take. With this distinction, execution of a program may be divided into *compartments*. Within a compartment, behaviour of the program mostly depends on general inputs. Compartments are separated from each other by complex checks on specific inputs. Random testing is therefore ideally suited to quickly test a wide variety of values for general inputs *within* compartments. On the other hand, Driller uses concolic testing to switch *between* compartments, by computing appropriate values for the specific inputs to satisfy the complex checks that separate them.

Static Program Verification

Concolic testing can also be combined with static program verification techniques, such as static analyses. Combining both techniques can help reduce the search space the tester has to traverse. For example, the static technique may prune many program paths that can statically be proven to be free of bugs [32].

Trimmer [47] is a hybrid concolic tester which employs static analysis for this exact purpose. Their static analysis infers a safety condition for bugs in the program: if a safety condition holds, the corresponding bug cannot be triggered. Trimmer then uses these safety conditions to instrument the program with assertions specifying these conditions. When the concolic tester executes the instrumented program, the conditions direct the tester towards bugs while helping prune paths in which bugs are known not to arise.

Wüstholz and Christakis [136] describe an approach for combining fuzzing with *online* static analysis to create a targeted hybrid fuzzer. Before the fuzzer adds a new input to its worklist, it launches a static analysis from this input to compute a path prefix for which all path suffixes will not pass through a target statement. The fuzzer can then use this prefix to avoid generating inputs which will follow this prefix.

3.6 Conclusion

We have presented an overview of the state of the art in concolic testing of distributed systems (Section 3.1), event-driven applications (Section 3.2), JavaScript applications (Section 3.3), and web servers (Section 3.4). We now conclude by presenting a list of concolic testers that were described in this chapter and which we deem most suitable for testing full-stack JavaScript web applications or most related to StackFul, and we motivate our selection. Afterwards, we evaluate these testers on the list of criteria for testing full-stack JavaScript web applications that were introduced in Section 2.4.

3.6.1 Identifying Concolic Testers for Full-stack JavaScript Web Applications

When identifying suitable testers, we are informed by the criteria for automatically testing full-stack JavaScript web applications outlined in section 2.4. Although the concolic testers described in this list may not meet all of these criteria, they should satisfy at least some of these. We therefore only consider testers which are, at minimum, capable of testing at least one side (client or server) of the full-stack JavaScript web application.

Our selection of most relevant testers consists of the following:

Apollo (Section 3.4.2) Apollo tests PHP server-side code with the aim of finding malformed HTML pages or other, generic PHP run-time errors. It does not explore the client, but simulates input received from the client to a limited degree, by modelling HTTP request parameters as symbolic inputs, thus giving it a limited degree of capability to reason over the client-server interaction.

ArtForm (Section 3.3.3) ArtForm focuses on testing form-based web applications. It can explore the behaviour of the client side of these applications, but does not explore the server side.

Cosette & JaVerT.Click (Section 3.2.1) Cosette is a powerful symbolic execution engine for JavaScript, capable of modelling a large part of the language. Its extension, JaVerT.Click, is even more powerful because of its modelling of event-driven code. Their extensive ability to test event-driven JavaScript code makes them a powerful tester for the client side of a full-stack JavaScript web application. Since JaVerT.Click is strictly more powerful than Cosette, we select JaVerT.Click to represent both tools.

CRAXWeb (Section 3.4.1) CRAXWeb finds security violations in server-side applications that can arise as a result of an HTTP request received from the client. Although it does not explore client-side code, it uses a web crawler on the client to find points where the client sends an HTTP request to the server. CRAXWeb is built on top of the S^{2}E framework. It therefore cannot analyse JavaScript code at the source level, but operates on compiled code.

Jalangi (Section 3.3.1) Jalangi is a record-replay tester for JavaScript: it first records an execution of the client by a user, and then replays this execution, using concolic testing to exercise alternate behaviours. Jalangi cannot by itself explore the event space of a client, but it can be used to explore a previously recorded event sequence.

KleeNet (Section 3.1.3) KleeNet is a dynamic symbolic execution engine for distributed systems, specifically, wireless sensor network applications. Although it is not designed to test web applications, its focus on modelling how the execution of one process may affect the execution of another process in a distributed system, while mitigating the state explosion problem in the system, is also relevant when testing a web application consisting of clients and servers.

Kudzu (Section 3.3.2) Kudzu tests JavaScript clients with the aim of detecting security vulnerabilities. It has limited capability for exploring a client's event space. Like the previous JavaScript testing tools, it does not simultaneously explore the server side of the web application, but still partially models communication between the two, by representing messages received via HTTP as symbolic inputs.

SymJS (Section 3.2.2) SymJS is capable of testing both standalone JavaScript applications and web clients. It places a particular emphasis on testing event-driven applications, by employing a read-write-conflict heuristic to direct the tester to generate sequences that are likely to expose different program behaviours. Although SymJS cannot be used to test the client and the server simultaneously, it is well equipped to explore the client side of such an application.

3.6.2 Evaluating Concolic Testers for Full-stack JavaScript Web Applications

We now evaluate these testers on the list of criteria for testing full-stack JavaScript web applications defined in Section 2.4. Our findings are listed in Table 3.1. For reference, the list of criteria on which we evaluate the testers is repeated in Table 3.2.

Table 3.1: The testers evaluated on the list of criteria identified in Section 2.4.

	①		②		③	④	
	A	B	A	B		A	B
Apollo	✗	✗	✗	✗	$\pm_1$	$\pm_2$	$\pm_3$
ArtForm	✓	✓	✓	✓	✗	✗	✗
CRAXWeb	$\pm_4$	✗	✗	✗	$\pm_1$	$\pm_2$	$\pm_{3,5}$
Jalangi	✓	✓	✓	✗	✗	✗	✗
Javert.Click	✓	✓	✓	$\pm_6$	✗	✗	✗
KleeNet	✗	✗	✗	$\pm_7$	✓	✓	✓
Kudzu	✓	✓	✓	$\pm_6$	$\pm_1$	$\pm_2$	$\pm_3$
SymJS	✓	✓	✓	✓	✗	✗	✗

[1] If only depending on HTTP communication
[2] One process explored, other processes simulated
[3] Simulates communication
[4] If compiled to supported bytecode
[5] Crawls client events
[6] No strategy for exploring event space
[7] Considers non-deterministic failure events

Apollo

Apollo is only capable of executing server-side PHP code and is therefore unsuited for testing client-side code: it cannot execute JavaScript code, is incapable of monitoring static or dynamic registration of events, it cannot explore the event space of a program and, although it can detect malformed HTML generated by the server, does not interact with the DOM at all.

Table 3.2: The criteria previously defined in Section 2.4.

①	A	Capable of testing sequential JavaScript code
	B	Models inputs from the DOM
②	A	Allows for dynamic (de)registration of event handlers
	B	Explores event space
③		Capable of finding composition-specific faults
④	A	Whole-program monitoring
	B	Observes communication between distributed processes

With regards to criteria 4.A and 4.B, Apollo is incapable of precisely modelling the interplay between frontend and backend in a full-stack JavaScript web application, since it can only observe the execution of the server and ignores the client. However, its ability to represent any HTTP communication received from the client as symbolic input lends it some capacity to model this interplay. Although it cannot observe or direct the behaviour of the client, it can over-approximate any outputs that are produced by the client, under any behaviour, by assuming all communication received from the client may take any value. This is possible even if the server is supposedly connected with multiple clients. However, this mocking of the execution of the client only serves to test the server process more completely, and Apollo remains incapable of actually finding any faults in the client.

We also give Apollo a partial score for criterion 3. On the one hand, it cannot infer the composition of processes necessary to exercise certain behaviour. However, if this composition would only have an effect on the values sent to the server over HTTP, then the appropriate composition may still be over-approximated, similar to criteria 4.A and 4.B.

ArtForm

ArtForm has adequate support for testing the client of a full-stack JavaScript web application and hence satisfies criteria 1-2. With regards to criterion 2.B, ArtForm inherits the strategies for exploring the event space for the Artemis tool, in which it is implemented.

The focus of ArtForm on testing only a single process leaves it incapable of reasoning over the interplay between frontend and backend of these applications. Furthermore, it does not have any ability to reason over the interplay between processes of a full-stack JavaScript web application: it cannot test any composition consisting of multiple processes. It only explores behaviour on the client and does not observe any communication received from the server. In fact, communication from the server is also not modelled, leaving ArtForm incapable of even simulating the behaviour of the entire web application.

CRAXWeb

CRAXWeb focuses on the testing of web servers and therefore does not aim to satisfy criteria 1-2. Because it uses the S^2E framework, which enables concolic testing of bytecode, it may satisfy criterion 1.A on the condition that JavaScript can be compiled to the appropriate bytecode format.

Since CRAXWeb only tests one individual process, it cannot reason over the interplay between processes of a full-stack JavaScript web application. However, much like Apollo, its ability to model HTTP communication received by the server from the client as symbolic inputs enables it to partially satisfy criteria 3-4. Like Apollo, mocking the execution of the client means that CRAXWeb cannot find client-side errors.

The imprecision caused by the over-approximation on the values sent by the client is partially mitigated by the fact that CRAXWeb also employs a web crawler to look for HTTP requests sent by the client and tests program paths on the server that start from the corresponding HTTP handler.

Jalangi

Jalangi's reliance on the record-replay technique limits its ability to test a full-stack JavaScript web client. It can execute JavaScript concretely and symbolically and can observe the dynamic registration of event handlers, but cannot by itself explore the event space of the client as it is limited to re-executing the event sequence that was recorded.

Like ArtForm, Jalangi only executes the client and ignores the server. Since it also does not model any communication received from the server, we conclude that Jalangi does not satisfy criteria 3-4.

JaVerT.Click

JaVerT.Click is a powerful symbolic execution engine for event-driven JavaScript. Since it is also capable of modelling the DOM, it satisfies criteria 1.A, 1.B, and 2.A. Regarding criterion 2.B, however, although JaVerT.Click describes semantics for (sequentially) executing individual event handlers, it does not specify a method for generating event sequences that efficiently explore the event space of the client. Furthermore, JaVerT.Click only executes the client and does not model or intercept communication from the server, leaving it incapable of satisfying criteria 3-4.

KleeNet

KleeNet tests wireless sensor network applications, and is capable of reasoning over the interplay between processes while observing their communication. Furthermore, KleeNet is not restricted to testing just one composition of processes, as users may configure the number and type of processes in the application to be tested. Hence, KleeNet can find composition-specific faults, if the appropriate composition is specified by the user.

Since KleeNet does not focus on testing web applications, it does not satisfy criteria 1-2.A. KleeNet is capable of traversing the event space of wireless sensor network applications by systematically injecting non-deterministic failure events, while reducing the size of the search space by preventing redundant forking of the nodes in the processes' symbolic execution trees. However, KleeNet does not address the criterion of dynamically registering or deregistering event handlers, as this challenge does not arise for non-deterministic failure events.

Kudzu

Kudzu is well-equipped to test full-stack JavaScript web clients: it can execute JavaScript code, draw symbolic input values from the DOM, and observe and trigger dynamically registered events. However, its strategy of generating random event sequences leaves it incapable of exploring the event space of the client efficiently.

Like many of the aforementioned JavaScript testing tools, Kudzu focuses on testing only a single, individual process, without reasoning over the behaviour of the entire distributed system. However, since Kudzu represents messages and values that are sent by the server via HTTP to the client as symbolic inputs, it is capable of over-approximating this behaviour in a manner similar to Apollo and CRAX-Web. Kudzu's mocking of the server's execution is only used to test a larger fraction of the client, however. Kudzu remains incapable of testing the server and reporting any bugs that may be present there.

SymJS

SymJS is a concolic tester that focuses on exploring the client of the full-stack JavaScript web application: it executes JavaScript code, handles dynamic registration of event handlers, uses a heuristic to intelligently explore the client's event space, and is capable of interacting with the DOM. However, like ArtForm, Jalangi and JaVerT.Click, it tests only the client and neglects the server process of the full-stack JavaScript web application. It furthermore does not model communication received from the server, so we conclude that it fails to satisfy criteria 3-4.

3.6.3 Overall Conclusion

After identifying several concolic testing tools described in this chapter and evaluating them on the criteria defined for testing full-stack JavaScript web applications, we conclude that no concolic tester yet exists which is fully equipped to test a full-stack JavaScript web application. In general, we can categorise the testers described here as testing either the client side (ArtForm, Jalangi, JaVerT.Click, Kudzu, and SymJS) or the server side (Apollo, KleeNet, and CRAXWeb), but there is no tool that tests both sides simultaneously.

Kudzu comes closest, because of its ability to test the client of such a system while modelling HTTP communication from the server as symbolic inputs, as this allows it to also test behaviours of the client that depend on the server communication specific messages or values. Even so, Kudzu merely models messages produced by the server, i.e., it mocks the execution of the server, without actually exploring the server. Hence, there is no concolic tester capable of exploring both the client and the server simultaneously.

4 A Foundation of Intra-process Concolic Testing

Having introduced the concept of concolic testing in Section 2.2, and having presented state-of-the-art concolic testers for various types of applications in Chapter 3, we now present *StackFul$_{INTRA}$*[1]. StackFul$_{\text{INTRA}}$ is a concolic tester for single-proces, event-driven JavaScript applications that access a Domain Object Model (DOM), such as web clients. This concolic tester will serve as a starting point for the development of a concolic tester capable of testing full-stack JavaScript web applications in Chapter 5 and Chapter 6.

Section 4.1 gives a high-level overview of the architecture of StackFul$_{\text{INTRA}}$, necessary for testing sequential JavaScript code. Sections 4.2 and 4.3 provide more detail on the individual components of StackFul$_{\text{INTRA}}$. Section 4.4 discusses how the procedure for concolic testing of sequential JavaScript code can be generalised to event-driven applications, by extending the concept of a symbolic execution tree to these types of applications. Section 4.5 provides a formal description of how StackFul$_{\text{INTRA}}$ performs concolic testing of single-process event-driven JavaScript applications. Section 4.6 concludes the chapter by evaluating StackFul$_{\text{INTRA}}$ on the criteria outlined in Section 2.4 for testing full-stack JavaScript web applications.

4.1 Overview of StackFul$_{\text{INTRA}}$

StackFul$_{\text{INTRA}}$ performs concolic testing on *individual* JavaScript applications, rather than full-stack JavaScript web applications. We coin this form of concolic testing on single JavaScript processes *intra-process* concolic testing, as testing takes place within a single process and execution does not cross process bound-

[1] `https://github.com/softwarelanguageslab/StackFul`

aries. Before discussing the technical details that enable StackFul$_{\text{INTRA}}$ to perform intra-process concolic testing, we give a broad overview of how StackFul$_{\text{INTRA}}$ tests sequential JavaScript programs.

4.1.1 Intra-process Concolic Testing with StackFul$_{\text{INTRA}}$

Concolic testing of any type of application necessitates the ability to monitor the execution of the application, since the tester must be able to construct symbolic constraints describing a path through he program, intervene in the program's execution, and generate appropriate values for program inputs arising through the program's execution. Broadly speaking, a concolic tester can use two approaches to accomplish these goals:

Modifying the execution engine The tester can be implemented as part of the engine that executes the JavaScript application, so that the tester can directly observe or intervene in the application's execution.

Instrumenting the code The source code or compiled bytecode of the application can be automatically instrumented, with the newly introduced code monitoring the execution of the program.

StackFul$_{\text{INTRA}}$ opts for using instrumentation to perform concolic testing on the application, as this has the advantage of not tying StackFul$_{\text{INTRA}}$ to a particular execution engine for running the code. This makes it possible to use StackFul$_{\text{INTRA}}$ both for testing standalone JavaScript code as well as JavaScript code that is embedded in an HTML web page. Depending on the type of application, StackFul$_{\text{INTRA}}$ will either spawn a Node.js[2] process to execute the instrumented standalone code, or an instance of jsdom[3] to execute the instrumented script. Node.js is a JavaScript runtime that can be used to execute event-driven JavaScript code. jsdom is a *headless* browser, i.e., a browser without a graphical user interface which nonetheless faithfully models the DOM of the web page.

4.1.2 Architecture of StackFul$_{\text{INTRA}}$

Figure 4.1 depicts the high-level architecture of StackFul$_{\text{INTRA}}$.

[2]https://nodejs.org/en/about
[3]https://github.com/jsdom/jsdom

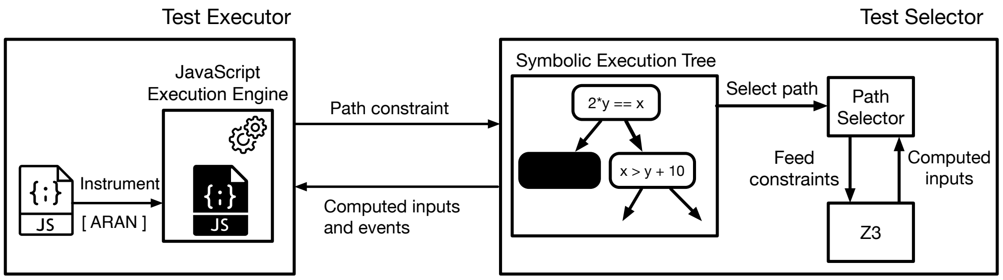


Figure 4.1: The architecture of StackFul$_{\text{INTRA}}$.

Broadly, StackFul$_{\text{INTRA}}$ consists of the following major architectural components (cf. Section 2.2.3):

Test Executor (Section 4.2) The test executor is responsible for performing concrete and symbolic execution on the JavaScript application under test while collecting the path constraint that represents the traversed program path. The test executor is further divided into different components for instrumenting the application, performing concrete and symbolic execution, and collecting the path constraint.

Test Selector (Section 4.3) The test selector is responsible for maintaining the symbolic execution tree (cf. Section 2.2.3), suggesting new program paths to explore in subsequent test runs, and computing appropriate values for the program inputs appearing in these paths.

The test executor and the test selector communicate with each other between test runs. The executor sends the path constraint of the program path that was explored in the most recent run to the test selector. Once the test selector has selected which program path to explore next, it computes values for the program inputs that feature in the selected paths, and sends these values to the test executor. When testing event-driven code, the test selector also sends the events that feature in the selected path to the test executor (cf. Section 4.4).

The text executor and test selector will be discussed in more detail in the following two sections.

4.2 The Test Executor

The test executor is responsible for executing the program under test, while intervening in its execution to generate appropriate values for the various program inputs and while observing the execution to collect the path constraint. To this end, the test executor employs the ARAN [34] framework for performing fine-grained instrumentation of the JavaScript source code to enable dynamic analysis of the application. In this section, we introduce ARAN and discuss how the executor uses it to monitor and intervene in the execution of the program under test.

4.2.1 Instrumenting JavaScript Code via Aran

ARAN [33] enables its users to specify program points, named *join points*, in selected source code files, that are then wrapped in calls to custom-defined functions, named *traps*. Join points include program expressions and statements, such as branch conditions in the control flow, function calls, dereferencing program variables etc. ARAN provides a total of 26 different traps that are executed when their corresponding join points are reached. Collectively, the set of traps is named the *advice*.

To illustrate, Listing 4.2 shows the result of using ARAN to instrument the code of Listing 4.1 (corresponding to lines 19–22 from Listing 2.3). ARAN transforms the `if` statement `if (right === 0)` into a series of calls to the `read`, `primitive`, `binary`, and `test` traps. These functions are used to inspect and possibly modify the default behaviour when respectively reading the variable `right`, accessing the literal value `0`, evaluating the binary expression `right === 0`, and handling the evaluated predicate expression of the `if` statement.

Aran allows for instrumenting both the source code under test as well as code that is loaded dynamically, e.g., via `eval`, `require`, or `<src>` tags. STACKFUL$_{\text{INTRA}}$ can therefore also perform concolic testing of dynamically loaded, yet fixed, code strings. STACKFUL$_{\text{INTRA}}$ expects the code strings to remain the same between test runs. Testing non-deterministic strings, such as user input, is therefore not supported Note that this is insufficient for evaluating non-deterministic strings such as user input, because STACKFUL$_{\text{INTRA}}$ expects the code string to remain the same between test runs. Evaluating arbitrary code strings may affect the execution of the program in entirely unpredictable ways, and hence result in the creation of symbolic execution subtrees that can take any shape. However, an empirical study has demonstrated that in the majority of cases, dynamic code execution is applied to constant strings [61], where StackFul$_{\text{INTRA}}$'s approach suffices.

```
if (right === 0) {
  throw new Error("Dividing by zero");
}
```

Listing 4.1: The code that is to be instrumented by Aran (lines 19–22 from Listing 2.3).

```
if (_.test(_.binary("===",
           _.read($right, "right"),
           _.primitive(0)))) {
  throw _.throw(_.construct(global.Error,
                            [_.primitive("Dividing by zero")]));
}
$result = _.write(_.binary("/",
                           _.read($left, "left"),
                           _.read($right, "right")),
                  "result");
```

Listing 4.2: The corresponding instrumented code generated by Aran.

Users of Aran can provide a custom implementation for each of the 26 traps, thereby allowing for fine-grained, dynamic analysis of a JavaScript application. More concretely, by overriding the implementation of these traps, Aran users can deploy a form of *shadow execution*[4] on the program.

4.2.2 Shadow Execution via Aran

Shadow execution [91] takes place alongside concrete execution but produces *shadow values*, values that describe the concrete values produced by the con-

[4]The term “shadow execution” is somewhat overloaded. In the context of this dissertation, we use this term to refer to the approach where a program is instrumented in order to compute a corresponding shadow value for every concrete value [34, 116]. The term should not be confused with the concept of *shadow symbolic execution* [65, 92], which can be used to compare two different versions of the same program.

crete execution. Shadow execution can be used to implement dynamic analyses, with the exact form taken by the shadow value dependent on the type of analysis that is to be implemented. In the case of StackFul$_{\text{INTRA}}$, these shadow values take the form of symbolic values that describe their associated concrete value.

To bind shadow values to their corresponding concrete values, StackFul$_{\text{INTRA}}$ overrides Aran's traps so that each expression in the program evaluates to a *tuple* of a concrete value and its associated symbolic value, rather than just to a concrete value. To illustrate, Listing 4.3 depicts a simple binary expression (Listing 4.3a), the instrumented version of this expression produced by Aran (Listing 4.3b), and a simplified representation of the implementation of the `primitive` and `binary` traps that appear in the instrumentation (Listing 4.3c).

Aran wraps each literal value, such as numbers, strings, booleans etc. in a call to the `primitive` trap. Binary expressions are instrumented so that the `binary` trap is called with the values for the left and right operands, as well as a string representation of the operator that is applied. As demonstrated in Listing 4.3c, the `primitive` trap ensures that each literal value evaluates to a tuple of a `concrete` and `symbolic` value, rather than just a concrete value alone. In the case of literal primitive values, the symbolic value either simply represents the concrete value lifted to the symbolic domain, or an empty symbolic value for those value types that are not supported by the symbolic execution, such as closures.

The `binary` trap expects the left and right operand to be instances of such tuples, with a concrete and symbolic part. This trap then itself returns a new tuple, where the concrete part equals the result of applying the operator on the left and right concrete operands and the symbolic part refers to a symbolic representation of the binary expression. Many other traps are similar to `binary` in that they take these tuples as arguments and produce new tuples.

```
1 + 2;
```

(a) A simple binary expression.

```
_.binary(_.primitive(1), "+", _.primitive(2))
```

(b) The expression instrumented with `primitive` and `binary` traps.

```
_.primitive = (primitive) => {
  concrete: primitive,
  symbolic: switch (typeof primitive) {
    case "number":  new SymInt(primitive);
    case "boolean": new SymBool(primitive);
    case "string":  new SymString(primitive);
    default:        new SymEmpty();
  }
};

_.binary = (operator, left, right) => {
  concrete: applyBinary(operator, left.concrete, right.concrete),
  symbolic: new SymBinaryExp(left.symbolic, operator, right.symbolic)
};
```

(c) The `primitive` and `binary` traps of ARAN, as implemented by STACKFUL$_{\text{INTRA}}$.

Listing 4.3: Shadow execution via the `primitive` and `binary` traps.

4.2.3 Using Shadow Values for Concolic Testing

Apart from producing shadow values, shadow execution must accomplish the following goals in order to perform concolic testing:

- Intercepting and generating program inputs
- Constructing symbolic expressions
- Updating the path constraint

We briefly discuss how ARAN's traps and shadow values enable STACKFUL$_{\text{INTRA}}$ to accomplish these goals.

Intercepting and Generating Program Inputs

Sources of non-deterministic program inputs, such as user inputs, environment inputs, system inputs, random values, or calls to external non-deterministic functions, must be identified. Identifying these sources is non-trivial, since they may arise from a variety of locations, such as retrieving user input, reading values from a file, command-line arguments etc. Apart from identifying these program inputs, StackFul$_{\text{INTRA}}$ must also symbolically mark such a value as a symbolic input value. Finally, StackFul$_{\text{INTRA}}$ must determine the appropriate concrete value for that program input. If the test selector computed a specific value for that input at the end of the previous test run, StackFul$_{\text{INTRA}}$ must use this value as the concrete value for the input. If no value was specified by the test selector, StackFul$_{\text{INTRA}}$ generates a random value of the appropriate type, with the type dependent on the source of the program input.

For one specific example of how program inputs are identified and processed, we turn to calls to a `randomInt`[5] function. Aran provides the `apply` trap, which fully replaces calls to functions. Thus, rather than calling the function directly, `apply` is called instead with as arguments the function to be applied, the list of arguments, and the *context* of the function, which specifies the object that is referred to when the `this` keyword is used. Listing 4.4 depicts a partial implementation of the `apply` trap to demonstrate how this trap can be used to identify and intercept program inputs. For the sake of brevity, the implementation of `nextProgramInput`, which returns either a concrete value precomputed by the test selector or a random value, has been elided.

```

_.apply = (func, context, arguments) => {
  if (func.concrete === randomInt) {
    return { concrete: nextProgramInput(),
             symbolic: new SymInputInt() };
  }
  ...
};
```

Listing 4.4: Part of the implementation of the `apply` trap.

[5]JavaScript does not have a built-in function that returns a random integer value. For the purpose of this dissertation, we therefore consider `randomInt` to be a hypothetical built-in function that returns a random integer value between `0` and `Number.MAX_SAFE_INTEGER`.

Constructing Symbolic Expressions

For concolic testing of applications, symbolically marking only the program inputs is insufficient to explore the entire program. Rather, *every* value that appears in the program should have a corresponding symbolic value so that the concolic tester is aware of the exact constraints that are placed on program inputs when a particular path is followed.

As a demonstration, consider Listing 4.5, where the branch condition depends on the inverse of the program input generated by `randomInt()`. Suppose that StackFulIntra randomly produced the value `42` for this input. The expression `-1 * x` then evaluates to `-42` and execution flows along the `else` branch of the `if` statement. The symbolic expression corresponding to `-42` should indicate that the value was produced as a result of taking the inverse of the program input, so that, when the test selector must compute an appropriate value for the program input to cause execution to flow along the `then` branch, it knows to compute a negative number.

```
let x = randomInt();
if (-1 * x > 1) {
  ... // elided
} else {
  ... // elided
}
```

Listing 4.5: An example demonstrating why symbolic expressions must be constructed.

Hence, every concrete value that is produced throughout the program's execution must be lifted to the symbolic domain by constructing symbolic expressions that capture the computations that were applied by the program in order to attain that value. StackFulIntra accomplishes this by relying on several Aran traps. An example of the implementation for some of these traps was shown in Listing 4.3c, which demonstrated how literal values and binary expressions are given an appropriate symbolic value.

The test executor fully supports the integer and boolean symbolic domains. It also models strings and supports some frequently used string operations, such as string indexing, checking for string equality, string concatenation, or checking for substrings. StackFulIntra does not employ an explicit memory model and is therefore incapable of reasoning over object constraints and arrays. Arrays hence only have a concrete representation. Indexing in an array always results in the executor concretising both the array index and the array element that is queried.

Updating the Path Constraint

Whenever a branch condition is encountered in the program's execution, the path constraint must be updated. In essence, this path constrained is formed by taking the logical conjunction of the symbolic expressions representing the predicate values of these conditions.

Aran provides the `test` trap, which wraps around the evaluated predicate of branch conditions, such as `if` statements and `while` loops. An example of this trap was included in Listing 4.2. StackFul$_{\text{INTRA}}$ therefore uses this trap to update the path constraint whenever a new branch condition is encountered. Listing 4.6 depicts a partial implementation of this trap.

```
let pathConstraint = new SymBool(true);

_.test = (predicate) => {
  pathConstraint = new SymBooleanConjunction(predicate.symbolic,
                                             pathConstraint);
  return predicate.concrete;
};
```

Listing 4.6: Part of the implementation of the `test` trap.

Modelling Library Functions

By default, StackFul$_{\text{INTRA}}$ does not instrument libraries as this would incur an additional overhead on the execution time of the program. Performing symbolic execution through the code of the library would also further increase the number of paths that should be explored by the tester. However, StackFul$_{\text{INTRA}}$ allows for constructing ad-hoc models of libraries that are consulted whenever a function of the library is called. For example, StackFul$_{\text{INTRA}}$ provides ad-hoc models of several jQuery[6] functions that are used to register event handlers, as well as functions for querying the content of a text input field.

[6]`https://jquery.com/`

4.3 The Test Selector

Whereas the test executor is mostly responsible for enabling individual test runs by monitoring and intervening in the execution of the program under test, the test selector (cf. Section 2.2.3) ensures that all paths through the program are explored systematically, over the course of multiple test runs.

To this end, the test selector must perform the following tasks:

- Maintain the symbolic execution tree
- Suggest new program paths for exploration
- Compute appropriate values for program inputs

We briefly elaborate on these tasks and discuss how they are accomplished by the test selector.

4.3.1 Maintaining the Symbolic Execution Tree

The test selector maintains the collection of program paths that have been explored through previous test runs. This collection takes the form of the symbolic execution tree, similar to the one depicted in Figure 2.4. At the end of each test run, the test executor sends the path constraint of the program path that was followed over the course of this run to the test selector, which adds it to the symbolic execution tree.

4.3.2 Suggesting Program Paths for Exploration

After having received the path constraint of the previous test run, the test selector must suggest a new program path to explore in the next test run. StackFul$_{\text{INTRA}}$ supports two exploration strategies for finding new program paths: a brute-force strategy and a more refined one. The former finds new program paths by traversing the tree in a breadth-first manner and by negating components of an existing path to create new paths. The latter exploration strategy is based on the one employed by the SymJS tester [72] (cf. Section 3.2.2) and keeps track of which variables are read from and written to by individual event handlers.

4.3.3 Computing Values for Program Inputs

When a new path has been selected, it is provided to an SMT solver that, if the path constraint is satisfiable, computes appropriate values for the symbolic input variables appearing in the constraint describing that path. If the path is unsatisfiable, a new path is selected. The test selector then forwards these results to the test executor, which will use them in the next test run.

StackFul$_{\text{INTRA}}$ employs the Z3 SMT solver [38, 15] for solving symbolic constraints. Specifically, StackFul$_{\text{INTRA}}$ uses Z3's theory of non-linear integer arithmetic and its theory of strings and sequences [142] to solve integer, boolean, and string formulae. As mentioned in Section 4.2.3, StackFul$_{\text{INTRA}}$ does not employ a memory model. It hence cannot create object constraints or reason symbolically over array operations.

4.4 Concolic Testing of Event-driven Applications

Sections 4.2 and 4.3 gave an overview of how StackFul$_{\text{INTRA}}$ performs concolic testing of sequential JavaScript programs that are not event-driven. However, as discussed in Section 2.3.2, both regular web applications and full-stack JavaScript web applications make heavy use of event-driven code. Consider the code snippet in Listing 4.7. Any automated tester that aims to systematically explore all possible paths through this program must not only consider the conditional branches in the program but also the sequences of events that may arise in the program's execution.

The exact behaviour of this code snippet depends on the order in which and the number of times both buttons are clicked, as well as the value of the random number that is generated each time `button1` is clicked. In fact, apart from the registration of these event handlers, no code is executed at all *until* a click event is triggered for either button. A naive, but complete, exploration of all possible program paths must therefore consider both the values of symbolic program inputs, such as the randomly generated number on line 3, as well as the sequence of events to be followed.

We will give a brief overview of how the test selector and the test executor have to be adapted to enable StackFul$_{\text{INTRA}}$ to test event-driven programs.

```
let counter = 0;
button1.addEventHandler("click", function (e) {
  if (randomInt() % 2 === 0) {
    console.log(counter);
  } else {
    counter = 0;
  }
}
button2.addEventHandler("click", function (e) {
  counter++;
}
```

Listing 4.7: An example of an event-driven program.

4.4.1 Supporting Events in the Test Selector

To systematically explore all paths through an event-driven program, it is necessary to add to the symbolic execution tree the paths through the various event handlers and to encode the decisions of which event was triggered at a point in the program's execution. One possible extension to the symbolic execution tree is shown in Figure 4.2, which depicts part of the symbolic execution tree for Listing 4.7.

Extending the Symbolic Execution Tree

In this extended symbolic execution tree, we distinguish between two different kinds of branching nodes. *Conditional branching nodes*, depicted as blue $i_i \% 2 = 0$ nodes in the figure, represent a point where control flow splits because of a branching condition such as an `if` statement or `while` loop. *Event branching nodes*, depicted as orange E_i nodes, represent a point in the execution where a new event can be triggered. These nodes determine which of the currently active event handlers should be executed next, i.e., for which currently active event handler the corresponding event should be triggered. The green `End` nodes indicate that program execution has ended.

In the case of JavaScript, event handlers are *atomic*, i.e., no two event handlers can be executed simultaneously: a new event handler can only be executed when the previous handler has been completed [133]. This implies that StackFulIntra should only consider triggering a new event when no event handler is

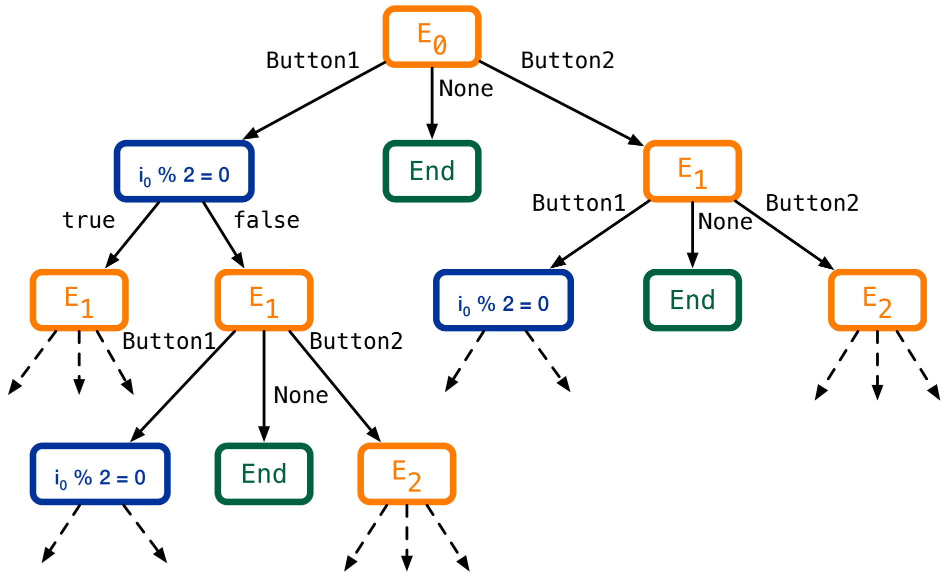


Figure 4.2: Part of the (infinite) symbolic tree produced by systematically testing the code of Listing 4.7.

currently being executed, thus conforming to the atomic execution of JavaScript event handlers. Event branching nodes are therefore only added at a point in the path in the symbolic tree where execution has already left an event handler.

Example of an Extended Symbolic Execution Tree

Consider again Listing 4.7. In this program, no code is executed outside of the event handlers, apart from registering these handlers via calls to `button.addEventHandler`, which does not involve branching of either kind. The first branching point of the program, i.e., the root node of the tree depicted as event E_0 in Figure 4.2, is therefore an event branching node representing the following choices:

- Invoking the first registered event handler (i.e., clicking `button1`).
- Invoking the second registered event handler (i.e., clicking `button2`).
- Not invoking any event handler at all, in which case the program terminates immediately.

If `button1` is clicked, the execution splits again into two possible paths:

- The expression `randomInt() % 2 === 0` evaluates to `true` and the `then` branch on line 4 is followed.
- The expression evaluates to `false` and the `else` branch on line 6 is followed.

Whatever button is clicked, once its corresponding event handler completes its execution, the previous three-fold choice is presented again for a potential second event E_1. In general, every event branching node is followed by $n+1$ child nodes, where n is the number of event handlers that are active at the corresponding point in the program's execution. An additional child node is included to account for the case where *no* event handler is invoked and execution of the program stops there. This number of child nodes hence increases or decreases depending on whether new event handlers are registered or existing event handlers are deregistered.

Note that this extension implies that the size of the tree in an event-driven program is infinite, as the choice of which event handler to execute next can be repeated indefinitely. Indeed, every possible event sequence corresponds to a unique path through the tree. Since an event sequence may in theory grow to an infinite length, there are an infinite number of unique paths through the tree, even if many of these program paths do not reveal any "new" program behaviour. Dashed arrows in the figure therefore indicate that the symbolic execution tree continues below these arrows but that subsequent nodes have been elided from the figure.

Concluding Notes on the Test Selector

Extending the symbolic execution tree with these event branching nodes largely allows the test selector to accomplish the first of its three tasks, to maintain the symbolic execution tree, in the context of event-driven programs as well. However, the test selector relies on the test executor to mark in its transmitted path constraints when an event handler was started and when execution left the handler. The test executor must additionally include the number of event handlers that are active when it starts executing a new event handler, so that the test selector is aware of how many alternative event handlers are available per event branching node.

The path exploration strategy of the test selector determines which program path of an event-driven program should be explored in subsequent runs. Some exploration strategies, such as the one employed by the SymJS tester wield sophisticated heuristics that select paths featuring specific event sequences. Other strategies, such as a breadth-first search strategy also implemented in StackFul$_{\text{INTRA}}$, mostly disregard the distinction between conditional and event branching nodes. Their decision to select a particular path is hence not affected by whether or not the JavaScript program under test features event-driven code. Regardless of the exploration strategy, the test selector must communicate the event sequence necessary for following a particular path through the symbolic execution tree to the test executor. This sequence simply corresponds to the sequence of events encoded by the event branching nodes appearing in the tree path.

The test selector does not need to be changed at all to complete its final task, computing values for program inputs, since these event branching nodes only represent a choice of which event handler to execute at which point in the execution. They are not associated with any specific program inputs.

4.4.2 Supporting Events in the Test Executor

As outlined in Section 4.2, the test executor must perform the following tasks: intercepting and generating program inputs, constructing symbolic expressions, and updating the path constraint. Adding support for event-driven programs to the executor does not require changes to the way in which it performs those existing tasks. However, it does necessitate two additional tasks: to intercept the registration and deregistration of new event handlers, and to follow the event sequence specified by the test selector for the current test run.

Intercepting the Registration and Deregistration of Event Handlers

Similar to how the test executor must identify and intercept in the generation of program inputs or the evaluation of non-deterministic expressions, the test executor must also intercept the registration and deregistration of event handlers. This information enables the test selector to understand which event handlers are available when an event branching node must be added to the symbolic execution tree.

Since event handlers can be registered and deregistered dynamically in JavaScript (cf. Section 2.3.2), the test executor must intercept these actions wherever they take place in the code. To this end, the test executor again relies on Aran traps. Specifically, the `apply` trap (cf. Section 4.2.3), which is executed whenever a function is called from instrumented code, consults a model enumerating the JavaScript functions known to register or deregister event handlers.

For example, when `apply` realises that the function to be called is the `addEventListener` method from the `EventTarget` interface[7], it performs the necessary steps to account for this event handler registration. As mentioned before, libraries are not instrumented by StackFul$_{\text{INTRA}}$ by default. Event handlers that are registered through library functions are missed by the test executor, unless a model for the library function has been provided. For example, StackFul$_{\text{INTRA}}$ defines an ad-hoc model of e.g., the `click` and `keypress` functions of jQuery, through which event handlers for these types of events can be registered. When a library function is called from instrumented code, the test executor may consult the model to determine whether an event handler has been registered.

Following Event Sequences

As mentioned before, when suggesting to the test executor a program path to follow, the test selector communicates both the values for the program inputs appearing in that path, as well as the sequence of events encoded in that path. The test executor follows the selected event sequence. As the test executor observed each event handler being registered along that path, it also understands which event should be generated in order to trigger the corresponding handler. In JavaScript, events, including user events such as button clicks or key presses, can be created and triggered programmatically. When an event is triggered, an event instance is created and passed as an argument to the registered event handler. These event instances typically include additional information, accessible via object fields, about the event, such as the pressed key code of a `KeyboardEvent`. As this information is considered a form of user input, reading these fields produces a symbolic input. For example, when a `MouseEvent` is triggered for a mouse click, the exact coordinates of the click are treated as symbolic inputs by the test executor as these coordinates may appear in conditional expressions in the event handler and affect the control flow of the program execution.

The most important invariant for the executor to uphold is the principle of atomicity for event handlers: the executor should not attempt to trigger the next event

[7] `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener`

while the previous handler is still being executed. To that end, the executor only creates and triggers the next event from the event sequence when no event handler is currently active.

The executor triggers the first event from the event sequence when all top-level code of the program, i.e., all code in the program that is located outside of event handlers, has been executed. Subsequent events are only triggered when execution has returned from the previous event handler. A test run terminates when all event handlers in the event sequence have been completed.

4.5 Formalisation of Intra-process Concolic Testing

To detail intra-process concolic testing in StackFul$_{\text{INTRA}}$ and facilitate replication, we provide a formal model of intra-process concolic testing for a minimalistic language that includes language features relevant to full-stack JavaScript web applications, such as event handlers and branching conditions. This model illustrates the workings of the test executor: the simultaneous concrete and symbolic execution process, the creation of path constraints, registration of event handlers, and creation and triggering of events while following a specified event sequence.

4.5.1 Overview of the Language

Figure 4.3 describes the syntax of the language. Its primitive values consist of integers, booleans, and closures. Atomic expressions are evaluated in a single evaluation step. They consist of constants (integers and booleans), lambdas, variables, `input` expressions which generate a non-deterministic number when evaluated, binary expressions, and `register` expressions for registering a closure as the handler for a given event or message type. The full set of expressions also includes `let` expressions and `if` expressions and function applications. `register` expressions correspond to calls to e.g., the `EventTarget.addEventListener` method from Section 4.4.2 and can be used to bind a closure to a given event type.

In the remainder of this section, we define a set of small-step evaluation rules that stipulate how StackFul$_{\text{INTRA}}$ would perform a single test run of an application written in this language. These evaluation rules operate on a CESK-machine-like [46] representation of the state of StackFul$_{\text{INTRA}}$'s test executor. Figure 4.4

defines the state space. Overlines in this figure denote sequences of elements. We use the : operator to conjoin individual elements to the head or the tail of sequences, and we use ϵ to represent an empty sequence.

A value in the language consists of a concrete and a symbolic component. The concrete component v_c is either an integer, a boolean, or a closure. The symbolic component v_s can either be a literal integer or boolean, a unique input variable labelled with an identifier (i.e., a symbol), the **empty** symbol (in case the concrete value cannot be represented symbolically), or a symbolic representation of a binary expression.

A state ς consists of: the current expression being evaluated, the current lexical environment which maps variables to addresses, the store which maps addresses to values, the continuation stack consisting entirely of continuation frames for `let` expressions, a path constraint, a (possibly empty) sequence of precomputed inputs, and a (possibly empty) sequence of preselected events. Note that the store maps to pairs of a concrete and a symbolic value, rather than just either a concrete or a symbolic value.

As explained in Section 4.4, in an event-driven program, the path constraint includes both conditional expressions and event handler selection nodes. The sequence of inputs represents the precomputed concrete values that will be assigned to the symbolic input variables, i.e., program inputs or sources of non-determinism, encountered during the test run. The event sequence represents the event handlers selected by StackFul$_{\text{INTRA}}$ to be executed consecutively. Whenever a particular handler has been completely executed, StackFul$_{\text{INTRA}}$ moves on to the next handler until all handlers have been executed.

We do not model how StackFul$_{\text{INTRA}}$ moves from one test run to the other. This involves the test selector choosing the sequence of inputs and of handlers to be invoked in the next run. Both sequences are computed by observing the path constraints that were collected during previous test runs. The exact mechanism for selecting them depends on the exploration strategy employed by the test selector. Instead of modelling this mechanism, we assume the existence of an external driver that provides both sequences.

The evaluation rules for StackFul$_{\text{INTRA}}$ are split in evaluation rules for atomic expressions and for the other expressions.

$$\begin{aligned}
c \in \mathit{Constants} &::= \; i \mid b \\
\mathit{lam} \in \mathit{Lam} &::= \; \lambda x.\, e \\
f, \mathit{ae} \in \mathit{Atom} &::= \; c \mid \mathit{lam} \mid x \mid \texttt{input} \mid \\
& \; \mathit{ae} \oplus \mathit{ae} \mid \texttt{register}\; m\; \mathit{lam} \\
x \in \mathit{Var} &::= \; \text{(a set of identifiers)} \\
m \in \mathit{HandlerType} &::= \; \text{(a set of identifiers)} \\
e \in \mathit{Exp} &::= \; \texttt{let}\; x\; \texttt{=}\; e\; \texttt{in}\; e \\
&\;\;\mid \; f\; \mathit{ae} \\
&\;\;\mid \; \mathit{ae} \\
&\;\;\mid \; \texttt{if}\; \mathit{ae}\; \texttt{then}\; e\; \texttt{else}\; e
\end{aligned}$$

Figure 4.3: The syntax of the minimalistic language.

$$\begin{aligned}
\mathit{clo} \in \mathit{Closure} &::= \; \mathbf{clo}(\mathit{lam}, \rho) \\
h \in \mathit{Handler} &::= \; \mathbf{handler}(\mathit{clo}, m) \\
\mathit{id} \in \mathbb{N} &::= \; \text{(an infinite set of identifiers)} \\
a \in \mathit{Addr} &::= \; \text{(an infinite set of addresses)} \\
v_c \in \mathcal{V}_c &::= \; i \mid b \mid \mathit{clo} \\
v_s \in \mathcal{V}_s &::= \; \mathbf{int}(i) \mid \mathbf{bool}(b) \mid \mathit{in}_{\mathit{id}} \mid \mathbf{empty} \mid v_s \oplus v_s \\
v \in \mathcal{V} \subseteq \mathcal{V}_c \times \mathcal{V}_s &::= \; \langle v_c, v_s \rangle \\
\rho \in \mathit{Env} &= \; \mathit{Var} \to \mathit{Addr} \\
\sigma \in \mathit{Store} &= \; \mathit{Addr} \to \mathcal{V} \\
\kappa \in \mathit{Kont} &::= \; \mathbf{letk}(a, e, \rho) \\
\kappa \in \mathit{KStack} &= \; \overline{\mathit{Kont}} \\
\mathit{ctrt} \in \mathit{Constraint} &::= \; v_s \mid h \\
\mathit{pc} \in \mathit{PC} &= \; \overline{\mathit{Constraint}} \\
\bar{\iota} \in \mathit{Inputs} &= \; \overline{\mathcal{V}_c} \\
\overline{h} \in \mathit{Handlers} &= \; \overline{\mathit{Handler}} \\
\varsigma \in \Sigma &= \; \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{KStack} \times \\
& \; \mathit{PC} \times \mathit{Inputs} \times \mathit{Handlers}
\end{aligned}$$

Figure 4.4: The state space of StackFul$_{\text{INTRA}}$.

4.5.2 Evaluating Atomic Expressions

Atomic expressions are evaluated via the atomic evaluation function $\mathcal{A}$, defined as:

$$\mathcal{A} = \mathit{Atom} \times \mathit{Env} \times \mathit{Store} \times \mathit{PC} \times \mathit{Inputs} \rightarrow \mathcal{V} \times \mathit{PC} \times \mathit{Inputs}$$

This evaluation function is listed in Figure 4.5. This function takes as input an atomic expression, an environment and a store, a path constraint and a sequence of precomputed inputs. It returns a value, the possibly updated path constraint, and the possibly updated input sequence.

When evaluating a constant, the atomic evaluation function produces a value of which the concrete component consists of the constant itself and the symbolic component corresponds to a *lifting* of the concrete value to the symbolic domain according to the lift function $\uparrow$. Lifting an integer i produces the symbolic value $\mathbf{int}(i)$, lifting a boolean produces $\mathbf{bool}(b)$. Evaluating a lambda produces a value of which the concrete component is a closure and the symbolic component is **empty** as closures are not represented symbolically. A variable is evaluated by retrieving its address from the environment and looking up this address in the store.

`input` expressions represent sources of non-determinism and program inputs, such as a call to `randomInt` or the key code of a `KeyboardEvent` event. An `input` expression evaluates to an input value. The concrete component of an input value may differ between test runs, while its symbolic component is a new symbolic input variable in_{id}, with id an identifier that uniquely represents this input in the current test run. The concrete component is either the first value v_c in the sequence of precomputed input values, or it is a purely random number if this sequence is empty. In case of the former, $\mathcal{A}$ returns the remainder $\bar{\iota}$ of the sequence alongside the non-deterministic value. In case of the latter, it returns the empty sequence ϵ.

A `register` expression is evaluated by wrapping the closure and the event type into a **handler** and appending this handler to the path constraint. The boolean `true` is returned to signal the closure successfully being registered. The registration is included as part of the path constraint because a handler might only be registered conditionally. Before the start of a new test run, the external driver must therefore consider the path constraint when determining which sequence of handlers to select. For the sake of rendering the evaluation rules simpler, the language assumes that a closure registered as an event handler that features exactly one parameter. When the handler is executed, it is passed an input value as argument. This argument represents non-deterministic inputs generated as a result of the handler being executed, similar to how StackFul$_{\textsc{intra}}$ treats e.g., the coordinates of a `MouseEvent` event or the key code of a `KeyboardEvent` as symbolic program inputs.

The atomic evaluation rule A-Binary stipulates that a binary expression is evaluated by successively applying $\mathcal{A}$ to the left and to the right operand. The concrete component of the resulting value is computed by simply applying the operator. In most cases, the symbolic component is simply a literal representation of this operator being applied to the symbolic values of the operands. Depending on the SMT solver that is used, however, some expressions involving non-linear binary operators (e.g., the modulo operator) cannot be represented symbolically as they cannot be solved by the SMT solver [21]. Concolic testers mitigate this problem via concretisation (cf. Section 2.2.3): the concrete result is lifted directly to the symbolic domain.

$$\begin{aligned}
\mathcal{A}(c, \rho, \sigma, pc, \bar{\iota}) &= \langle\langle c, \uparrow c\rangle, pc, \bar{\iota}\rangle \\
\mathcal{A}(\lambda x.\ e, \rho, \sigma, pc, \bar{\iota}) &= \langle\langle \mathbf{clo}(\lambda x.\ e,\ \rho), \mathbf{empty}\rangle, pc, \bar{\iota}\rangle \\
\mathcal{A}(x, \rho, \sigma, pc, \bar{\iota}) &= \langle \sigma(\rho(x)), pc, \bar{\iota}\rangle \\
\mathcal{A}(\texttt{input}, \rho, \sigma, pc, v_c : \bar{\iota}) &= \langle\langle v_c, in_{id}\rangle, pc, \bar{\iota}\rangle \\
&\quad \text{With } id \text{ a new, unique identifier} \\
\mathcal{A}(\texttt{input}, \rho, \sigma, pc, \epsilon) &= \langle\langle i_r, in_{id}\rangle, pc, \epsilon\rangle \\
&\quad \text{With } i_r \text{ a random number and} \\
&\quad id \text{ a new, unique identifier} \\
\mathcal{A}(\texttt{register}\ e\ lam, \rho, \sigma, pc, \bar{\iota}) &= \langle\langle \texttt{true}, \mathbf{bool(true)}\rangle, pc : \mathbf{handler}(\mathbf{clo}(lam, \rho), e), \bar{\iota}\rangle
\end{aligned}$$

A-Binary

$$\frac{\mathcal{A}(ae_1, \rho, \sigma, pc, \bar{\iota}) = \langle\langle v_{c1}, v_{s1}\rangle, pc', \bar{\iota}'\rangle \qquad \mathcal{A}((ae_2, \rho, \sigma, pc', \bar{\iota}') = \langle\langle v_{c2}, v_{s2}\rangle, pc'', \bar{\iota}''\rangle}{\mathcal{A}(ae_1 \oplus ae_2), \rho, \sigma, \bar{\iota}) \ = \ \langle\langle v_{c1} \oplus v_{c2}, v_s\rangle, pc'', \bar{\iota}''\rangle}$$

With v_s equal to $v_{s1} \oplus v_{s2}$ if $v_{s1} \oplus v_{s2}$ can be modelled symbolically,

or equal to $\uparrow (v_{c1} \oplus v_{c2})$ if it cannot.

Figure 4.5: The atomic evaluation function $\mathcal{A}$.

4.5.3 Evaluating Non-atomic Expressions

Figure 4.5 lists the evaluation rules for non-atomic expressions. These evaluation rules take a state ς as input and return either a **next** if the test run can proceed with a new state, a **fail** if the test run terminated because an error was encountered, or a **stop** if the test run was stopped because StackFul$_{\text{INTRA}}$ executed all preselected handlers. In case of a **fail**, evaluation stops and the current path constraint is returned to indicate under which circumstances an error arises in the program.

Rule E-Let stipulates that `let` expressions are evaluated by allocating a new, unique address and creating a new environment ρ' where the variable is bound to this address. StackFul$_{\text{INTRA}}$ pushes a new **letk** continuation frame to evaluate the body once the value to be bound has been evaluated. The evaluation rule employs the auxiliary function alloc, not modelled here, to generate a unique address. Note that `let` expressions are the only type of expressions that result in a frame being pushed onto the continuation stack.

E-PopContinuation concerns atomic expressions. It uses the evaluation function $\mathcal{A}$ to evaluate this expression to a value v, and it considers the continuation stack to determine where this value must flow to. Since the stack consists entirely of **letk** frames, StackFul$_{\text{INTRA}}$ proceeds by assigning a, saved in the frame, to v in the store and continuing with the `let` body expression.

E-Application describes how function applications of the form $f ae$ are evaluated. As both the function and the argument expression are atomic expressions, they are evaluated with the atomic evaluation function $\mathcal{A}$, with the function expression required to evaluate to a closure. The closure's environment is extended with a binding from the parameter to a new, unique address a, and the store is extended with a binding from a to the argument value. Evaluation then proceeds with the function's body.

E-IfTrue demonstrates how the path constraint is updated when a branch condition is encountered. It specifies that if the predicate of an `if` expression evaluates to `true`, the path constraint is updated by appending the symbolic component of the predicate value to the path constraint.

E-IfFalse is similar to E-IfTrue. This rule describes the case where the predicate expression evaluates to `false`. Like E-IfTrue, the path constraint is extended, but with the *negation* of the symbolic component of the predicate value.

E-Let

$$\frac{a = \textsc{alloc}() \qquad \rho' = \rho[x \mapsto a]}{\langle \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2, \rho, \sigma, \kappa, pc, \bar{\iota}, \overline{h}\rangle \to \mathbf{next}(\langle e_1, \rho, \sigma, \mathbf{letk}(a, e_2, \rho') : \kappa\rangle, pc, \bar{\iota}, \overline{h}\rangle)}$$

E-PopContinuation

$$\frac{\mathcal{A}(ae, \rho, \sigma, pc, \bar{\iota}) = \langle v, pc', \bar{\iota}'\rangle}{\langle ae, \rho, \sigma, \mathbf{letk}(a, e', \rho') : \kappa, pc, \bar{\iota}, \overline{h}\rangle \to \mathbf{next}(\langle e', \rho', \sigma[a \mapsto v], \kappa, pc', \bar{\iota}', \overline{h}\rangle)}$$

E-Application

$$\frac{\begin{array}{c}\mathcal{A}(f, \rho, \sigma, pc, \bar{\iota}) = \langle\langle \mathbf{clo}(\lambda x.\ e', \rho'), \texttt{empty}\rangle, pc', \bar{\iota}'\rangle \\ \mathcal{A}(ae, \rho, \sigma, pc', \bar{\iota}') = \langle v, pc'', \bar{\iota}''\rangle \qquad a = \textsc{alloc}()\end{array}}{\langle f\ ae, \rho, \sigma, \kappa, pc, \bar{\iota}, \overline{h}\rangle \to \mathbf{next}(\langle e', \rho'[x \mapsto a], \sigma[a \mapsto v], \kappa, pc'', \bar{\iota}'', \overline{h}\rangle)}$$

E-IfTrue

$$\frac{\mathcal{A}(ae, \rho, \sigma, pc, \bar{\iota}) = \langle\langle \texttt{true}, v_s\rangle, pc', \bar{\iota}'\rangle}{\langle \texttt{if}\ ae\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \rho, \sigma, \kappa, pc, \bar{\iota}, \overline{h}\rangle \to \mathbf{next}(\langle e_1, \rho, \sigma, \kappa, pc' : v_s, \bar{\iota}', \overline{h}\rangle)}$$

E-IfFalse

$$\frac{\mathcal{A}(ae, \rho, \sigma, pc, \bar{\iota}) = \langle\langle \texttt{false}, v_s\rangle, pc', \bar{\iota}'\rangle}{\langle \texttt{if}\ ae\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \rho, \sigma, \kappa, pc, \bar{\iota}, \overline{h}\rangle \to \mathbf{next}(\langle e_2, \rho, \sigma, \kappa, pc' : \neg v_s, \bar{\iota}', \overline{h}\rangle)}$$

E-HandlerWithInput

$$\frac{a = \textsc{alloc}() \qquad \rho'' = \rho'[x \mapsto a] \qquad \sigma' = \sigma[a \mapsto \langle v_c, in_{id}\rangle]}{\langle ae, \rho, \sigma, \epsilon, pc, v_c : \bar{\iota}, \mathbf{handler}(\mathbf{clo}(\lambda x.\ e', \rho'), m') : \overline{h}\rangle \to \mathbf{next}(\langle e', \rho'', \sigma, \epsilon, pc, \bar{\iota}, \overline{h}\rangle)}$$

With id a new, unique identifier.

E-HandlerNoInput

$$\frac{a = \textsc{alloc}() \qquad \rho'' = \rho'[x \mapsto a] \qquad \sigma' = \sigma[a \mapsto \langle i_r, in_{id}\rangle]}{\langle ae, \rho, \sigma, \epsilon, pc, \epsilon, \mathbf{handler}(\mathbf{clo}(\lambda x.\ e', \rho'), m') : \overline{h}\rangle \to \mathbf{next}(\langle e', \rho'', \sigma, \epsilon, pc, \bar{\iota}, \overline{h}\rangle)}$$

With id a new, unique identifier and i_r a random number

E-Error

$$\frac{\mathcal{A}(ae, \rho, \sigma, pc, \bar{\imath}) = \lightning}{\langle ae, \rho, \sigma, \kappa, pc, \bar{\imath}, \overline{h}\rangle \rightarrow \textbf{fail}(pc)}$$

E-NoMoreHandlers

$$\frac{\mathcal{A}(ae, \rho, \sigma, pc, \bar{\imath}) = \langle v, pc', \bar{\imath}'\rangle}{\langle ae, \rho, \sigma, \epsilon, pc, \bar{\imath}, \epsilon\rangle \rightarrow \textbf{stop}}$$

Figure 4.5: Evaluating non-atomic expressions.

E-HandlerWithInput describes that a preselected handler is invoked when StackFul$_{\text{INTRA}}$ has reached an atomic expression with an empty continuation stack. As this indicates that no code is now being executed, StackFul$_{\text{INTRA}}$ can start executing the next event handler from its event sequence. It does so by popping the first handler from the sequence and proceeding to evaluate the body of the closure. Since the parameter of the closure is a non-deterministic input variable, it is assigned the first value v_c from the sequence of precomputed inputs, if this sequence is non-empty, similar to how the atomic evaluation function evaluated `input` expressions.

E-HandlerNoInput is similar to E-HandlerWithInput but describes the case where a new handler is mocked and no precomputed input is available. Similar to how `input`-expressions are evaluated when no precomputed inputs are available, a value-pair is generated where the concrete component is i_r a random value, and the symbolic component is a new, unique symbolic input variable.

E-Error describes how StackFul$_{\text{INTRA}}$ handles the case where atomic evaluation results in an error, represented here as the symbol $\lightning$, e.g., because an undefined variable was read or a binary operator was applied to incompatible types. StackFul$_{\text{INTRA}}$ wraps the current path constraint in a **fail** and terminates the test run.

E-NoMoreHandlers stipulates that a test run terminates when StackFul$_{\text{INTRA}}$ reaches an atomic expression with an empty continuation stack and an empty event sequence. Assuming the evaluation of this atomic expression does not result in an error (cf. E-Error), the test run terminates with a **stop**.

4.6 Conclusion

In this chapter, we presented how StackFul$_{\text{INTRA}}$ performs intra-process concolic testing of JavaScript applications, i.e., concolic testing applied to individual

JavaScript processes. We summarise the topics discussed in this chapter and conclude by evaluating this intra-process tester on the criteria outlined in Section 2.4 for testing full-stack JavaScript web applications.

4.6.1 Summary

We gave an overview of the architecture of StackFul$_{\text{INTRA}}$, with a test executor and a test selector.

The test executor uses the Aran framework to instrument a program's source code. This program instrumentation allows for performing shadow execution, so that symbolic execution can take place alongside concrete execution. Specifically, the executor intercepts and generates program inputs, constructs symbolic expressions, and updates the path constraint. In the case of an event-driven JavaScript application, it also intercepts the dynamic registration and deregistration of event handlers, and ensures that a prescribed event sequence is followed without violating the atomicity of JavaScript event handlers. At the end of each test run, the test executor communicates the path constraint observed for this run to the test selector and receives from the selector the set of program inputs and the event sequence to follow in the next run.

The test selector maintains the symbolic execution tree, selects from the tree a new program path to be explored in the next test run, and computes appropriate values for the program inputs appearing in this path. Event-driven programs can be supported by extending the representation of a symbolic execution tree with event branching nodes.

To facilitate replication, we presented a formal model of how StackFul$_{\text{INTRA}}$ performs single test runs on programs, driven by an external driver which suggests program inputs and event sequences to be followed by the model. These programs are written in a small language featuring numbers, booleans, and closures, and which further allows for the generation of non-deterministic program inputs, and for registering and invoking event handlers. This language, although minimalistic, therefore features all elements that are essential to model the concolic testing of event-driven JavaScript programs.

4.6.2 Concluding Remarks

The intra-process tester described in this chapter allows for concolic testing of individual JavaScript processes. Although it is therefore by design insufficient to enable testing of full-stack JavaScript web applications, Table 4.1 nonetheless evaluates the intra-process StackFul$_{\text{INTRA}}$ tester on the set of criteria identified for testing full-stack JavaScript web applications outlined in Section 2.4.

Table 4.1: The intra-process StackFul$_{\text{INTRA}}$ tester evaluated on the list of criteria identified in Section 2.4.

Criterion		Description	Satisfied?
①	A	Capable of testing sequential JavaScript code	✓
	B	Models inputs from the DOM	✓
②	A	Allows for dynamic (de)registration of event handlers	✓
	B	Explores event space	±
③		Capable of finding composition-specific faults	✗
④	A	Whole-program monitoring	✗
	B	Observes communication between distributed processes	✗

Criterion 1

Criterion 1 relates to whether the tester is capable of testing web clients. This criterion is subdivided into criteria 1.A, which requires the tester to be capable of testing web clients, and 1.B, which requires the tester to consider a web application's domain object model (DOM) for program inputs.

Criterion 1.A is satisfied by construction. StackFul$_{\text{INTRA}}$'s test executor, which is responsible for executing the JavaScript program under test, relies on the Aran framework for instrumenting the program. Aran's traps enable implementing shadow execution of the program, which in turn allows for monitoring of and intervening in the execution of the program at a fine-grained level, without the tester itself having to model the various intricacies of JavaScript's semantics. StackFul$_{\text{INTRA}}$ is therefore capable of executing JavaScript code.

Criterion 1.B requires the tester to consider the DOM as a source of non-deterministic program inputs so that all possible user interactions on a web page may be tested automatically. For example, DOM elements associated with user inputs, such as clicking a button, writing content into a text field, interacting with an HTML canvas etc. should all be considered as a form of program input, as these actions may affect the execution of the web application.

This criterion is also satisfied by means of StackFul$_{\text{INTRA}}$'s test executor. When the JavaScript code queries the state of an HTML element, this query is intercepted by one of the corresponding Aran traps, which in turn enables the test executor to intervene and generate an appropriate value for the resulting program input. Of course, the test executor must rely on an extensive model of which JavaScript queries should result in which kinds of program inputs. However, the construction of such a model is an engineering challenge rather than a fundamental obstacle.

Criterion 2

Criterion 2 relates to testing event-driven programs. It requires the tester to be capable of exploring the event space of the program and to allow for the dynamic, i.e., conditional, registration and deregistration of event handlers. Section 4.4 discussed how our test executor and test selector support testing of event-driven programs. StackFul$_{\text{INTRA}}$'s test executor is capable of intercepting dynamic registration and deregistration of event handlers by leveraging the appropriate Aran traps, e.g., the `apply` trap, and by building a model of JavaScript libraries and standard functions describing which methods perform a registration or deregistration. The first subcriterion is therefore satisfied.

The second subcriterion, whether the event space of the program can be explored efficiently, depends on the exploration strategy used by the test selector. StackFul$_{\text{INTRA}}$'s test selector implements two search strategies: a brute-force breadth-first strategy and another strategy based on the one employed by the SymJS tester [72]. Although both strategies enable exploration of a program's event space, the *efficiency* of the exploration, i.e., how quickly the tester is capable of traversing this space, strongly depends on the strategy.

Criteria 3-4

Criterion 3 and subcriteria 4.A and 4.B all relate to how a full-stack JavaScript web application is composed of the interconnected server process and one or more client processes. Criterion 3 demands that the tester is capable of finding program errors which may only arise in a specific composition of processes. Criteria 4.A and 4.B require the tester to be capable of monitoring the execution of all processes simultaneously, while taking into account the communication and interplay between the various processes. Since the tester described in this chapter only tests individual JavaScript processes, without taking into consideration how this process may be part of a larger application, none of these criteria are satisfied.

Overall Conclusion

The intra-process StackFul$_{\text{INTRA}}$ tester satisfies three of the seven criteria. One criterion, questioning whether the event space of the application can be explored efficiently, is only partially satisfied for now, as it relies on the test selector employing an efficient exploration strategy. The remaining three criteria relate to how a concolic tester can be constructed which tests a collection of interconnected JavaScript processes. In Chapter 5, we will show how StackFul$_{\text{INTRA}}$ can be extended so that it is able to satisfy these criteria as well.

5 Inter-process Concolic Testing

The previous chapter introduced StackFul$_{\text{Intra}}$, a concolic tester for event-driven, but single-process, JavaScript. As demonstrated in Section 4.6, StackFul$_{\text{Intra}}$ satisfies three of our seven criteria for performing automated testing on full-stack JavaScript web applications: i) it is capable of testing sequential JavaScript code, ii) it can access the DOM and is therefore capable of modelling user inputs, as well as system inputs and environment inputs, iii) and its ability to observe the dynamic registration and deregistration of event handlers, combined with the possibility of creating and dispatching events, allows StackFul$_{\text{Intra}}$ to explore an application's event space.

One criterion, regarding the exploration of a web application's event space, was partially satisfied. Although StackFul$_{\text{Intra}}$ is capable of traversing the event space, the *efficiency* at which it does so is strongly dependent on the exploration strategy employed by the test selector. The extent to which this criterion is satisfied will be reconsidered in Chapter 6.

The remaining three criteria relate to how a full-stack JavaScript web application is composed of multiple communicating processes. These criteria consist of: i) whether the tester is capable of finding composition-specific faults, ii) whether the tester can monitor the execution of the constituent processes of the full-stack JavaScript web application simultaneously, and iii) whether the tester can observe the communication between these processes. As StackFul$_{\text{Intra}}$ observes only the execution of one individual process, it trivially fails to meet these criteria.

In this chapter, we describe an extension of the previous tester, named *StackFul*$_{\textit{Inter}}$[1] because of its ability to perform *inter-process* concolic testing, that satisfies these last three criteria. We define inter-process concolic testing as concolic testing that takes place over multiple communicating processes simultaneously, with the tester observing the communication between all processes, and having execution paths cross process boundaries. We introduce inter-process testing by means of a use case in which we revisit the Calculator full-stack

[1]`https://github.com/softwarelanguageslab/StackFul`

JavaScript web application presented in Section 2.1.2. In this use case, we show how inter-process testers such as StackFul$_{\text{INTER}}$ enable differentiating between server errors that are of high or low importance, based on whether the error is reachable from a given client in a specific client-server configuration or not. We hypothesise that developers may prioritise fixing server-side errors that are reachable from the client over server-side errors that are unreachable from the client. It should be noted that server-side errors that are unreachable from a given client may still occur when an end-user accesses the server via a different client, or when they bypass the client altogether. We therefore do not intend for this use-case to distinguish between server-side errors that should be fixed and server-side errors that may remain unpatched. However, we believe that developers may find this a useful categorisation to decide which bugs should be patched first.

After introducing the use case, Section 5.2 gives an overview of inter-process testing and describes the architecture of StackFul$_{\text{INTER}}$. Section 5.3 then describes how this use case is tackled by StackFul$_{\text{INTER}}$. To further detail this use case, Section 5.4 extends the formal description of intra-process concolic testing (cf. Section 4.5) to the domain of inter-process testing. Section 5.5 evaluates the use case on a suite of full-stack JavaScript web applications. Section 5.6 concludes the chapter by evaluating StackFul$_{\text{INTER}}$ on the set of seven criteria for performing automated testing of full-stack JavaScript web applications.

5.1 Motivating the Need for Inter-process Concolic Testing

In this section, we present a use case for an inter-process concolic tester and describe how StackFul$_{\text{INTER}}$ realises this scenario, as opposed to how intra-process testers such as StackFul$_{\text{INTRA}}$ would handle it. This use case focuses on differentiating server-side errors in full-stack JavaScript web applications based on whether or not an error is reachable from a client in a specific client-server configuration.

5.1.1 Revisiting the Calculator Application

Consider again the full-stack Calculator application presented in Section 2.1.2. Before sending the arithmetic expression to the server, the client checks whether the expression is a valid arithmetic expression with a recognised operator (line 14 of Listing 2.1). If it is not, the client instead displays a warning to the user. The server performs an identical check on the payload of the messages that it receives (line 23 of Listing 2.3) and throws an error if the expression's operator cannot be resolved to a known arithmetic operator. The server also performs an additional check on whether the expression corresponds to a division by zero (line 20). For this last check, there is no equivalent client-side validation.

Traditional, intra-process concolic-testers such as StackFul$_{\text{INTRA}}$ would have to test both sides of the application in isolation from each other, since these testers only test individual processes. When applied to the client, StackFul$_{\text{INTRA}}$ should exercise the event handlers of all buttons and the callback for receiving the server message containing the result of the computation (lines 22–25 of Listing 2.1). 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 separate from the client process under test, in which case the client may send concrete messages to the server and expect real messages back. In either case, StackFul$_{\text{INTRA}}$ should be able to achieve 100 % line coverage for the client.

When testing the server, StackFul$_{\text{INTRA}}$ could again opt to exercise the computation request callback (lines 8–28 of Listing 2.3) by mocking client messages. However, as the server is being tested in isolation from the client, StackFul$_{\text{INTRA}}$ 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. In practice, when combining this server code with the previously described client code, it is clear that the error on line 23 is unreachable, as this particular client ensures that only messages using any of the four valid operators are sent to the server. On the other hand, the division-by-zero error on line 20 is still reachable on the server, as this client does not check for this error. We therefore say that the division-by-zero error is a *high-priority* error, while the invalid operator error is a *low-priority* error. Providing such a classification might increase developers' confidence in the testing tool, as evidenced for static bug detectors [105, 97, 69].

However, in the case of intra-process testing, the tester only considers one of either the client or the server processes. To test these processes, it must mock user-triggered events and inter-process messages. In the case of the latter though, no constraint can be placed on these messages as the tester does not know where these messages would come from nor how they would be produced. It is therefore unable to distinguish between the importance of both errors.

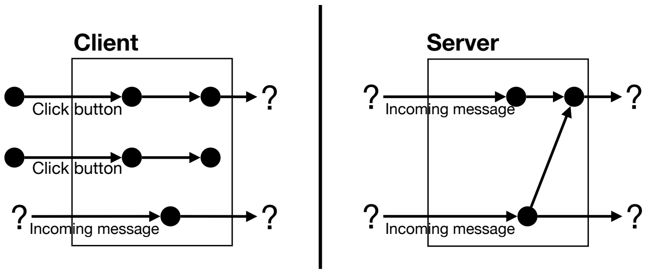


Figure 5.1: Intra-process testing of a client and a server in isolation from each other

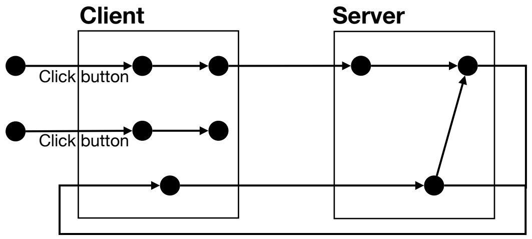


Figure 5.2: Inter-process testing of a particular client-server configuration

Differentiating between both kinds of errors is only possible by using an inter-process tester such as StackFul$_{\text{INTER}}$ which can simultaneously consider the server-side constraints that result in a high-priority error from a message handler, and the client-side constraint that cause the client to send this message in the first place. As an inter-process tester, StackFul$_{\text{INTER}}$ observes messages that are sent by this particular client as a result of user-triggered events and can track these messages as they reach the server, making StackFul$_{\text{INTER}}$ aware of *how* these messages were produced. In the case of this Calculator application, messages with an invalid operator would never be sent to the server but would be halted by the client on line 14 of Listing 2.1. When combined with traditional, intra-process testing of the server in isolation, StackFul$_{\text{INTER}}$ can realise that the division-by-zero is a high-priority error, as it can be triggered by any user exercising this client. The error involving the unknown operator is a low-priority error as it can only be triggered in combination with a different client, or by circumventing the client altogether. This issue is visualised for a generic full-stack JavaScript web application in Figure 5.1 and Figure 5.2.

5.2 Overview of Inter-process Concolic Testing

In order to perform inter-process concolic testing of a full-stack JavaScript web application, the concolic testing procedure must be extended to span across all constituent processes of the application. As an example of how an inter-process concolic tester would test a full-stack JavaScript web application, consider Listing 5.1. This application consists of a client and a server process which com-

```
1  // Client side
2  const socket = io();
3  let x = randomInt();
4  if (x > 10) {
5    socket.emit("msg", x + 1);
6  }
7  // Server side
8  ... // Create and connect socket
9  socket.on("msg", function (data) {
10   let y = randomInt();
11   if (y === 42 && data < 15) {
12     throw new Error();
13   }
14 });
```

Listing 5.1: A simple full-stack JavaScript web application.

municate with each other via websockets (lines 2 and 9). The client generates a symbolic input, in the form of a random value, that is assigned to the variable `x` (line 3). The client conditionally sends a message with *name* `msg` and with *payload* `x + 1` to the server (line 5). Upon receiving the message (line 9), the server generates a symbolic input of its own (line 10) and conditionally throws an uncaught error (line 12). An inter-process concolic tester should be capable of finding and reporting this server-side error.

5.2.1 Requirements for Inter-process Concolic Testing

We identify three requirements that must be satisfied in order to enable inter-process concolic testing of the code presented in Listing 5.1 and, more generally, to distinguish between high-priority and low-priority errors in the CALCULATOR application:

Concolic testing on the process composition as a whole An inter-process concolic tester must perform simultaneous concrete and symbolic execution on every process in the application.

Global path constraint The tester must construct a *global path constraint* that includes constraints that were observed in an arbitrary application process.

Modelling message payload The tester must observe and model communication between processes, and provide a symbolic representation of the payload of the messages sent.

Concolic Testing on Both Processes

The server-side error is only triggered if both the client-side condition guarding the message send operation and the server-side condition guarding the uncaught error evaluate to `true`. In order to explore both consequent branches, the random value assigned to `x` on the client side and the random value assigned to `y` on the server side must attain specific values. It is therefore necessary to collect symbolic constraints on these conditions in both processes of the application, so that appropriate values can be computed by feeding these constraints to the test selector.

Constructing a Global Path Constraint

Performing concolic testing on both processes would lead the tester to construct the symbolic constraint $x_0 > 10$ for the condition on line 4 and $y_0 = 42 \wedge x_0 + 1 < 15$ for the condition on line 11, where x_0 and y_0 refer to the symbolic inputs assigned to `x` and `y` respectively. As both constraints place restrictions on the concrete values that x_0 can assume in order to reach their respective consequent branches, the concolic tester cannot consider these constraints in isolation from each other. If the tester were to ignore the server-side constraint, it might compute a concrete value for x_0 that is larger than `15`, and execution would hence not reach the error on line 12. Likewise, if the first constraint were to be disregarded, the tester might compute a value smaller than `10`, which would result in the message not being sent in the first place so that the error likewise would not be reached.

A concolic tester executing the client-server composition as a whole would therefore have to construct the global path constraint $x_0 > 10 \wedge y_0 = 42 \wedge x_0 + 1 < 15$, which is formed by taking the conjunction of the client-side constraints and server-side constraints. Using this global path constraint, the tester can trivially find solutions for x_0 and y_0 that steer the execution towards the error.

Modelling Message Payload

In order to construct the global path constraint, when sending the message from the client to the server, the symbolic representation of `data`, i.e., $x_0 + 1$, must be sent along with its concrete value. If this symbolic expression is not sent, concolic testing on the server side would be forced to fall back to either concretisation or representing `data`'s symbolic expression as a symbolic input variable. In both cases, server-side constraints that involve `data` would then incorrectly reflect which constraints must be placed on the client-side input x_0.

5.2.2 Architecture of StackFul$_{\text{INTER}}$

In order to satisfy these three requirements, StackFul$_{\text{INTER}}$ employs a dynamic analysis that spans all constituent processes of the full-stack JavaScript web application. To this end, StackFul$_{\text{INTER}}$ uses Aran-Remote [35], an analysis platform that enables deploying a single instance of a dynamic analysis across several processes. Aran-Remote inserts the same traps around join points in the source code of the processes as those inserted by Aran (cf. Section 4.2.1). However, the trap functions introduced into the instrumented code only contain a stub implementation. When a trap is called, the stub serialises the trap arguments and sends a synchronous request to the *central analysis process*, which in turn executes the actual trap. This central analysis process is shared between all instrumented processes and atomically handles all advice requests in the order in which they are received. When the trap has finished executing on the central analysis process, the return value is serialised and sent back to the requesting process, which then resumes its execution. Because of the frequent synchronous communication, an unfortunate side-effect is that the central analysis process may introduce a rather large performance overhead.

The specification of which client and server processes should be launched to test the full-stack JavaScript web application is configured by the user of StackFul$_{\text{INTER}}$. This user specifies the types of the processes (i.e., their source code) and the number of instances (i.e., their multiplicity) that should be created of this process type. At startup time, StackFul$_{\text{INTER}}$ follows this specification and sequentially launches each process.

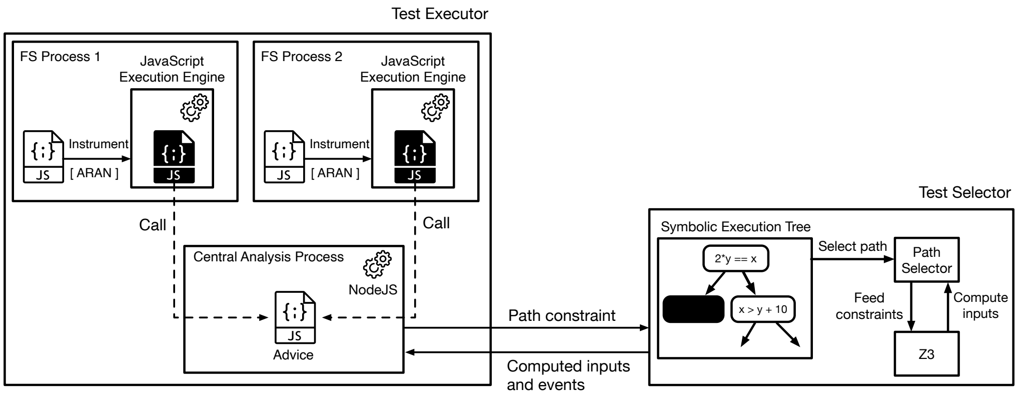


Figure 5.3: The architecture of StackFul$_{\text{INTER}}$.

Figure 5.3 illustrates the architecture of StackFul$_{\text{INTER}}$. A full-stack JavaScript web application consists of one or more JavaScript processes. The source code for each of these processes is first instrumented via Aran, after which the instrumented code is executed by a JavaScript engine, such as a browser or the Node.js runtime. The central analysis process itself is executed as a Node.js process. It sets up a proxy to listen for incoming requests from one of the traps in the instrumented full-stack processes and processes each request atomically. The central analysis process and the full-stack JavaScript web application processes together form the test executor (cf. Section 4.2).

Similar to the test executor described for StackFul$_{\text{INTRA}}$, the executor is responsible for constructing symbolic expressions and collecting symbolic constraints across all full-stack processes. This makes it possible for the central analysis process to construct the global path constraint, as it can store one central instance of a path constraint and concatenate it whenever it observes a new, individual constraint. Note that the central analysis process does not take race conditions between processes into account. However, race conditions only affect the *order* in which constraints are added to the global path constraint.

The test selector for StackFul$_{\text{INTER}}$ is identical to the one employed in StackFul$_{\text{INTRA}}$ (cf. Section 4.3), although the symbolic variable inputs that appear in the global path constraint may originate from several different full-stack processes.

By having the central analysis process observe the concrete and symbolic execution of the full-stack processes via the advice API, as well as having the central analysis process construct a global path constraint, the first and second requirements for performing inter-process concolic testing are met by StackFul$_{\text{INTER}}$.

The third requirement, modelling the messages' payload, is achieved by having the test executor intercept all websocket traffic. When the test executor observes that a message is sent via a websocket, e.g., via `socket.emit`, to another process that is part of the full-stack JavaScript web application, the executor intercepts the message send operation. It then ensures that for each value in the payload of the message, both a concrete and a symbolic representation of this payload is transmitted.

The test executor likewise intercepts the registration of message handlers, via e.g., `socket.on`, and registers a separate handler instead. This handler unpacks the message, and for each value in the payload, binds the symbolic representation of this value to the concrete representation. Afterwards, the original message handler is called and execution resumes as normal.

5.3 Prioritising Server-side Reachability

As described in Section 5.1, inter-process testing would enable distinguishing between high-priority and low-priority server-side errors, because of its ability to simultaneously consider which server-side constraints lead a message handler to trigger an error, and which client-side constraints result in the client sending this message in the first place. StackFul$_{\text{INTER}}$ allows for distinguishing between both kinds of errors. Rather than performing inter-process testing from the start, which would quickly run into the state explosion problem due to the increased lengths of the path constraints, StackFul$_{\text{INTER}}$ operates in two phases. In the first phase StackFul$_{\text{INTER}}$ automatically tests the server side of the web application in isolation over the course of several test runs, i.e., it performs intra-process testing of the server process. In the second phase, StackFul$_{\text{INTER}}$ performs inter-process testing of a particular client-server configuration to attempt to exercise the client in such a way that the previously discovered server errors are triggered. Once testing terminates, any error that was reproduced via this particular client-server configuration is labelled a high-priority error, while errors that could not be reproduced are labelled low-priority.

We proceed by describing both phases.

5.3.1 Intra-process Testing Phase

In both the intra-process and inter-process testing phases, StackFul$_{\text{INTER}}$ employs the architecture outlined in Section 5.2.2. StackFul$_{\text{INTER}}$ therefore has a central analysis process available during the intra-process phase as well, even though during this phase the central analysis only observes the execution of the server process.

The central analysis process continuously monitors all information that is relevant for determining which program paths are available on the process under test. This includes information about the symbolic conditions the analysis observes and the various message listeners that are registered.

During the intra-process testing phase on the server, incoming messages must be mocked to test the server's message handling code. StackFul$_{\text{INTER}}$ hence treats the server as an event-driven application, where the reception of a mocked message is considered an ordinary event, so intra-process testing of the server proceeds in a manner similar to that described in Section 4.4. To avoid any race conditions, the analysis triggers the reception of a newly mocked message only when it has determined that the message handler for the previous message has terminated. The payload of these messages is treated as consisting entirely of symbolic input variables.

The collected path constraints hence consist of a sequence of incoming, mocked messages as well as the true or false outcomes of conditional branches which may or may not depend on the values of the message's symbolic input variables. As the central analysis process has information available on the symbolic branch conditions as well as which message listeners have been registered, the analysis knows which paths are available and can communicate this information to the test selector. Similar to StackFul$_{\text{INTRA}}$, StackFul$_{\text{INTER}}$ steers the execution of the server so it follows the path that was prescribed by the test selector by controlling the sources of non-deterministic execution, such as e.g., `randomInt` and the sequence and payload of the messages that must be mocked. Whenever StackFul$_{\text{INTER}}$ encounters an error during the execution of the application, it stores the full path constraint that led to this error, including the message handlers for which a message was mocked.

5.3.2 Inter-process Testing Phase

Initially, the inter-process testing phase is similar to the first phase in that the central analysis process and the test selector work in tandem to explore the client process of the web application over the course of multiple test runs. As the client is generally event-driven, the path constraints collected by the analysis are once again a combination of symbolic branch conditions and triggered event sequences. As in the first phase, the analysis process intervenes in the execution of the client both when a non-deterministic value is generated, and to ensure that the prescribed event sequence is followed. The inter-process phase can be divided in three stages: exploring the client-side code until StackFul$_{\text{INTER}}$ observes a message being sent to the server, determining whether this message could carry a payload that triggers a server error, and starting a new test run in which the previous run is replayed but the payload of the message is changed.

We describe these stages using Listing 5.2 as an example. In this program, a client generates two random integers (lines 4–5) and sends a message `msg` to the server if the first number is greater than `10`. The server performs an additional check. If the `x` field in the message's payload is greater than `15` and the `y` field equals `1`, an error is thrown. Suppose that this error was discovered during the intra-process testing phase. The path constraint leading to this error would be $\text{data}.y = 1 \ \wedge\ \text{data}.x > 15$, with the additional information that a `msg` message was mocked to exercise the `msg` handler.

5.3.3 Exploring the Client

This stage of the inter-process phase is similar to the exploration of the server during the intra-process phase. The main goal of this stage is to discover, over the course of several test runs, a program path that results in a message being sent. Whenever this happens, the central analysis process checks whether this message can result in a previously discovered error to be reached on the server side. If any such errors are found, StackFul$_{\text{INTER}}$ proceeds to the next stage. If a test run finishes without a message being sent, StackFul$_{\text{INTER}}$ starts a new test run and explores a different path.

In the case of Listing 5.2, one branch of the event handler results in a message being sent. Suppose that at some point a test run generates the random integers

```
1 // Client side
2 const socket = io();
3 document.addEventHandler("click", function (e) {
4   let x = randomInt();
5   let y = randomInt();
6   if (x > 10) {
7     socket.emit("msg", {x: x + 1, y: y });
8   }
9 });
10 // Server side
11 ... // Create and connect socket
12 socket.on("msg", function (data) {
13   if (data.y === 1 && data.x > 15) {
14     throw new Error();
15   }
16 });
```

Listing 5.2: Client and server code of an event-driven full-stack JavaScript web application.

`11` and `3` for `x` and `y` respectively so that the message on line 7 is sent, with a payload of `12` for `data.x` and `3` for `data.y`. StackFul$_{\text{INTER}}$ remembers the path constraint for the current test run, $x > 10$, and proceeds to the next stage.

5.3.4 Considering the Message's Payload

The goal of this stage is to determine whether the message being sent can carry a concrete payload such that both the current, client-side path constraint is satisfied (so that the message will indeed be sent) and the server-side path constraint corresponding to that error is also satisfied (so that this payload will lead to the expected error). If this is not possible, the current test run proceeds, but StackFul$_{\text{INTER}}$ returns to the previous stage until it observes another message being sent. If it is possible, StackFul$_{\text{INTER}}$ continues with this stage.

Returning to our example, StackFul$_{\text{INTER}}$ has observed a message `msg` being sent to the server. A message of this type could result in the error on line 14 being triggered. At this point, StackFul$_{\text{INTER}}$ must employ the message payload to *synchronise* the client-side path constraint with the server-side path constraint that led to the error. The server path, data.$y = 1 \, \wedge$ data.$x \, > 15$ employed two symbolic input variables, data.x and data.y, to mock the message payload. The client path defines its own constraints on parts of the payload, namely $x > 10$. Both paths must be joined together by explicitly including constraints that equate all mocking symbolic inputs with their actual symbolic values: $x + 1 = \text{data}.x \wedge y = \text{data}.y$. The full, synchronised path constraint is therefore $x > 10 \wedge x + 1 = \text{data}.x \wedge y = \text{data}.y \wedge \text{data}.y = 1 \wedge \text{data}.x > 15$. Although the current, concrete values for `x` and `y`, `11` and `3` respectively, will not result in the server error being thrown, StackFul$_{\text{INTER}}$ feeds the full path constraint to an SMT solver to find values for `x` and `y`, e.g., `18` and `1`, so that the full constraint is satisfied. StackFul$_{\text{INTER}}$ then proceeds to the final stage.

5.3.5 Replaying the Test Run

StackFul$_{\text{INTER}}$ starts a new test run in which it uses the values that were computed for the appropriate symbolic variables in the last stage to verify whether this new run succeeds in reproducing the expected error. If it does, the error is marked as a *high-priority* error, since it has been proven that the client can be exercised in such a way that an error is produced on the server side. StackFul$_{\text{INTER}}$ afterwards returns to the first stage to resume exploring the client, until its test budget is exhausted. Any remaining unmarked errors are then automatically classified as *low-priority* errors.

In the case of Listing 5.2, it can be observed that the previously computed values for `x` and `y` indeed result in the server error being re-triggered.

Note that it is generally impossible to *prove* that an error is of only low priority. Given an infinite number of program paths available on the client, StackFul$_{\text{INTER}}$ can only ascertain that none of the program paths that were explored within a particular budget led to the rediscovery of a previously discovered error.

5.4 Formal Description of Server-side Error Prioritisation

We extend the formal model of intra-process concolic testing previously presented in Section 4.5 to the domain of inter-process testing in order to detail how StackFul$_{\text{INTER}}$ distinguishes between high-priority and low-priority server-side errors. In this section, we only describe the changes that have to be made to this previous model.

$$\begin{aligned} e \in \mathit{Exp} ::= \quad & ... \\ | \quad & \texttt{send}\ m\ ae \end{aligned}$$

Figure 5.4: Syntax of a `send` expression.

The formal model previously presented in Section 4.5 detailed how concrete and symbolic execution were performed over the course of a single test run. It did not describe a mechanism for selecting a sequence of inputs or event or message handlers were to be triggered in a specific test run, as such a mechanism relies on the exploration strategy employed by the test selector. Instead of modelling such a mechanism, the model assumed the existence of an external driver which provides both sequences.

We likewise assume the existence of such a driver for StackFul$_{\text{INTER}}$, which also decides when to switch between the intra-process and the inter-process testing phases. The state space of StackFul$_{\text{INTER}}$ is identical to that of StackFul$_{\text{INTRA}}$ (cf. Figure 4.4). However, we assume that the external driver holds a collection of errors θ defined as a map from handler types to a collection of path constraints.

5.4.1 Extending the Syntax of the Minimal Language

We extend the minimal language which served as the basis for this formal model (cf. Figure 4.3) with the ability to send messages with payloads from one process to another. We assume that full-stack JavaScript web applications in this language consist of just two processes. There is hence no need to specify to *which* process a message should be sent. The syntax for these `send` expressions is shown in Figure 5.4. A `send` expression is used to send a message of type m to the other process in the full-stack JavaScript web application. The message is received by the corresponding message handler that was registered via the previously introduced `register` expression. The payload of the message is the value that the atomic expression ae evaluates to.

5.4.2 Extending the Evaluation Rules for Non-atomic Expressions

Recall that each evaluation rule for a non-atomic expression resulted in three distinct types of values:

- **stop**, indicating that a test run should be stopped because all preselected event or message handlers have been executed.
- **fail**(pc), indicating that the tester detected an error in the application, and that the path constraint pc leads to this error.
- **next**(ς), indicating that evaluation of the expression was successful, and that evaluation should proceed with the state ς.

We update the **fail** so it also includes the type of the message handler that was being executed. A **fail** hence becomes **fail**(pc, m), where pc is the path constraint that leads to an error in an application and m is the type of the message handler whose execution led to the error being triggered.

The evaluation rules for atomic expressions in StackFul$_{\text{INTER}}$ do not have to be changed. The evaluation rules for non-atomic expressions in StackFul$_{\text{INTER}}$ are also almost identical to those in StackFul$_{\text{INTRA}}$. Only two changes have to be made to the evaluation rules for atomic expressions. These changes are listed in Figure 5.5.

E-ErrorInter replaces evaluation rule E-Error (cf. Figure 4.5). Recall that E-Error described how StackFul$_{\text{INTRA}}$ handled detection of an error during evaluation of atomic expressions. E-ErrorInter is largely similar to this rule, but employs the auxiliary function GetCurrentHandlerType, not modelled here, to find the type of the preselected handler currently being executed. StackFul$_{\text{INTER}}$ wraps the current path constraint as well as the handler type m of the handler currently being mocked in a **fail**. The external driver holds an error collection θ_{server} for the server, which is a map from handler types to a set of path constraints leading to server errors. The external driver then extends the set of path constraints for the handler type m with the path constraint pc.

E-SendSatisfiable models the second stage of inter-process testing (5.3.4) and describes the evaluation of a `send` expression on the client side. We assume that the external driver provides the error collection θ_{server} that was gathered after intra-process testing of the server was completed.

E-ErrorInter

$$\frac{\mathcal{A}(ae, \rho, \sigma, pc, \bar{\iota}) = \lightning \qquad m = \textsc{GetCurrentHandlerType}()}{\langle ae, \rho, \sigma, \kappa, pc, \bar{\iota}, \overline{h}\rangle \rightarrow \mathbf{fail}(pc, m)}$$

E-SendSatisfiable

$$\frac{\begin{array}{c}\mathcal{A}(ae, \rho, \sigma, pc, \bar{\iota}) = \langle\langle v_c, v_s\rangle, pc', \bar{\iota}'\rangle \\ \exists pc'' \in \theta_{server}(m) : \quad pc' : (in_{id} = v_s) : pc'' \text{ is satisfiable}\end{array}}{\langle \texttt{send}\, m\ ae, \rho, \sigma, \kappa, pc, \bar{\iota},\ \overline{h}\rangle \rightarrow \mathbf{stop}}$$

Where in_{id} is the symbolic input variable assigned to

the mocked message handler that caused the error.

Figure 5.5: Evaluating non-atomic expressions in StackFul$_{\text{INTER}}$.

The goal is to find a server error, previously reported via the E-ErrorInter rule during intra-process testing, which would become reachable because of the message send. The error can only be reachable if both the current path constraint pc' is true (so that this message is sent in the first place) and the server-side path constraint pc'' is also true (so that upon arriving, the message leads to the error). Furthermore, messages carry payloads, represented as the atomic expression ae in this rule. While this payload was mocked with a symbolic input variable during the intra-process testing phase, it now receives an actual value by applying the atomic evaluation function on the payload. This value must explicitly be bound in the path constraint to the original input variable[2] of the mocked message handler whose execution led to the error during the intra-process phase. Any check performed by the server on the mocked input would have appeared as a constraint in pc'', while any check that appeared on the message payload would have appeared in pc'. By explicitly equating the mocked input with the actual value, any check applied by one side also becomes applicable to the other. StackFul$_{\text{INTER}}$ therefore joins pc' and pc'' together via the equality $in_{id} = v_s$. If the concatenation of both path constraints is satisfiable, the server error can be reproduced via this message send, so the test run stops.

[2]We do not model how the original input variable in_{id} is found, but it appears in the path constraint pc''.

Table 5.1: Characteristics of the web applications considered in our study.

Benchmark	LOC	# of Branches	WMC	# of Message and Event Handlers
CALCULATOR	126	16	18	16
CHAT	288	39	27	9
GAME OF LIFE	214	64	32	10
SIMPLE CHAT	45	4	6	2
SLACK MOCKUP	662	75	20	13
TOHACKS	144	22	20	5
TOTEMS	145	9	14	5
WHITEBOARD	126	16	14	3

5.5 Evaluation

In this section, we evaluate the inter-process testing approach. We perform this evaluation in the context of the inter-process testing use case, where we employed inter-process testing to distinguish between high-priority and low-priority server errors (cf. Section 5.3). We measure the extent to which inter-process testing improves precision, by comparing the inter-process and intra-process (baseline) approaches to concolic testing of full-stack JavaScript web applications. Specifically, we measure how capable the two-phase approach to inter-process testing is at discerning low-priority from high-priority errors.

5.5.1 Overview of the Evaluation

We define the following research questions:

- **RQ1:** How many high-priority server errors are correctly classified by STACKFUL$_{\text{INTER}}$ as being of high priority?
- **RQ2:** Are there any instances of STACKFUL$_{\text{INTER}}$ incorrectly classifying low-priority server errors as being of high priority?
- **RQ3:** How many test runs of inter-process testing does STACKFUL$_{\text{INTER}}$ require to reproduce a high-priority error?

The context of the study consists of eight different programs (Calculator, Chat, Game of Life, Slack Mockup, Simple Chat, TOHacks, Totems, and Whiteboard). The source code of these applications has been made available at `https://github.com/softwarelanguageslab/StackFul`. The Calculator program is the example full-stack JavaScript application introduced in Section 2.1.2. Whiteboard and Chat are demo applications[3] for the Socket.IO library. Game of Life, TOHacks, Totems, and Simple Chat were retrieved from a software gallery[4] featuring applications built with the Socket.IO library. Slack Mockup is a project that mimics some of the functionalities of the Slack communication platform. Apart from the Calculator and Simple Chat applications, which only feature a client state, all programs maintain both a client and a server state, which is manipulated by the event and message handlers. Table 5.1 reports the main characteristics of each web application: the number of lines of code, the number of branches in the code, the Weighted Methods per Class (WMC) metric, and the number of message and event handlers that are registered in the application. During our evaluation, we spawn for every benchmark one instance of the server process and one instance of a client process that connects with this server.

To evaluate the inter-process approach, we introduced additional synthetic pairs of equivalent checks in both the client and server for the Game of Life, Simple Chat, Totems, and Whiteboard programs. These equivalent checks, similar to the client and server checks featured in the Calculator example, conform to the following pattern: the client checks whether a condition on a value holds, and, if it does, sends a message pertaining to this value to the server. The server then checks a condition that is equivalent to the one performed by the client. Examples of such checks include whether or not the coordinates of a mouse click fall within or outside a certain frame, or whether certain words in a chat application should be censored instead of being broadcasted. These checks effectively create regions of dead code on the server as, given this particular client-server configuration, these conditions should never be false. Note that, since these checks depend on the value of the payload, static dead code checks do not suffice for discovering these regions.

Having introduced these checks, we automatically and randomly inject synthetic `"ERROR: INJECTED SERVER ERROR"` faults in the server of the resulting programs. This because the collected benchmark programs contain few faults by themselves. Every `if` branch or `case` branch in the program has the same probability of being

[3]`https://github.com/socketio/socket.io/tree/master/examples`
[4]`https://devpost.com/software/built-with/socket-io`

injected by the fault injection process. Faults injected in the server that fall within the aforementioned dead code region are considered *low-priority* errors, while faults injected outside of these regions are *high-priority* errors. Listing 5.3, which corresponds to lines 15–16 of Listing 2.3, depicts an `"ERROR: INJECTED SERVER ERROR"` fault being injected on line 2.

```
1 case "+": {
2   "ERROR: INJECTED SERVER ERROR";
3   {
4     result = left + right;
5     break;
6   }
7 case "-": result = left - right; break;
```

Listing 5.3: Part of the server code for one of the two Calculator variants, with an injected synthetic fault.

To increase the generalisability of our study, we applied the fault injection process twice on each benchmark application, except for the Simple Chat application because of the application's small size. This fault injection thus resulted in two variants being produced for most applications. Both variants of a particular application differ only in the number and location of the injected faults.

To answer the research questions, we ran StackFul$_{\text{Inter}}$ on all 15 resulting variants. For the first phase, StackFul$_{\text{Inter}}$ was allocated a test budget of 250 test runs to ensure most server errors (both high-priority and low-priority errors) were found. In the second phase, a test budget of 500 test runs was allocated to rediscover these errors.

Table 5.2 reports on the classification of high-priority and low-priority errors for each application. For both categories, the table specifies the total number of errors of that category that were injected into the server of that application (*# of Faults*), and the number of these errors that StackFul$_{\text{Inter}}$ correctly labelled as belonging to that category (*Correctly Classified*). We manually verified for each application whether the classification generated by StackFul$_{\text{Inter}}$ was correct.

Specifications

The test executor, implemented in TypeScript, uses Node.js v18.7.0, while the test selector, implemented in Scala, uses Java SE Runtime Environment

Table 5.2: Classification in high-priority and low-priority errors for each variant.

Application	High-Priority		Low-Priority	
	# of Faults	Correctly Classified	# of Faults	Correctly Classified
Calculator I	5	3	2	2
Calculator II	6	6	1	1
Chat I	0	0	1	1
Chat II	1	0	2	2
Game of Life I	1	1	1	1
Game of Life II	2	1	2	1
Simple Chat	0	0	1	1
Slack Mockup I	3	0	3	2
Slack Mockup II	4	1	1	1
TOHacks I	3	3	0	0
TOHacks II	2	2	0	0
Totems I	2	2	0	0
Totems II	0	0	2	2
Whiteboard I	2	2	2	2
Whiteboard II	5	5	3	3

build 19.0.1+10-21, configured to run with a maximum heap size of 4GB. StackFul$_{\text{Inter}}$ was executed on a 2.8 GHz Quad-Core Intel Core i7 CPU, with 16GB of 2133 MHz LPDDR3 RAM, running macOS 13.2. As an SMT solver, StackFul$_{\text{Inter}}$ uses Z3, version 4.8.5 - 64bit.

We answer the three research questions over the following sections. In Section 5.5.6, we go into more details on the answers that are offered here.

5.5.2 RQ1: Correct Classification of High-Priority Errors

The first research question asks how many high-priority server errors are correctly classified by StackFul$_{\text{Inter}}$ as being of high priority. All of the applications combined contain a total of 36 high-priority server errors. StackFul$_{\text{Inter}}$ is able to correctly classify 26 of these. Calculator I, Chat II, Game of Life II and both versions of Slack Mockup contain misclassified high-priority errors. For the first three applications, these errors were uncovered during the intra-process phase but were not reproduced during the inter-process phase, even though they were reachable from the client. In effect, StackFul$_{\text{INTER}}$ incorrectly labelled these high-priority errors as being of low priority. For the Slack Mockup variants, these errors were also not uncovered during the intra-process phase.

In all cases, these errors are located along program paths that can only be reached from the client when several events are triggered in a particular order, and when additional client-side conditions are met. This phenomenon was particularly outspoken in Slack Mockup, as some of the event handlers involved are not registered upfront, but only dynamically under certain conditions. StackFul$_{\text{INTER}}$ therefore first needs to discover a program path where the appropriate handlers are registered before this path can be explored further by triggering the corresponding sequence of events. In general, it is conceivable that StackFul$_{\text{INTER}}$ would have correctly classified the errors for all of these applications given a larger test budget. Practical concerns, such as the long (up to five hours for the budget of $250 + 500 = 750$ combined test runs) running time of each benchmark and the memory footprint of Aran-Remote's approach to dynamic analysis of distributed applications, necessitate a limit on the test budget.

We conclude that, for the given applications, StackFul$_{\text{INTER}}$ is able to classify the majority of high-priority errors correctly without exhausting its test budget.

5.5.3 RQ2: Misclassifying Low-Priority Errors as High-Priority Errors

The second research question asks whether there are any instances of StackFul$_{\text{INTER}}$ incorrectly classifying low-priority server errors as being of high priority. As can be observed from Table 5.2, the applications total 21 low-priority server errors. StackFul$_{\text{INTER}}$ misclassifies only two of these, one for the Game of Life II application and one for the Slack Mockup I application. In both cases, these errors were only misclassified because they were not discovered during the intra-process testing phase in the first place. However, StackFul$_{\text{INTER}}$ did not reproduce any of the low-priority errors during its inter-process testing phase. There were hence no instances of low-priority errors being misclassified as being of high priority.

5.5.4 RQ3: Inter-process Test Runs

The third research question asks how many test runs of inter-process testing StackFul$_{\text{INTER}}$ requires to reproduce a high-priority error. Table 5.3 specifies for

Table 5.3: The number of inter-process test runs StackFul$_{\text{INTER}}$ required to classify each of the high-priority errors it found.

Error	# of Test Runs	Error	# of Test Runs
Calculator I: Error 1	33	Slack Mockup II: Error 1	240
Calculator I: Error 2	33	Slack Mockup II: Error 2	>500
Calculator I: Error 3	449	Slack Mockup II: Error 3	>500
Calculator I: Error 4	>500	Slack Mockup II: Error 4	>500
Calculator I: Error 5	>500	TOHacks I: Error 1	1
Calculator II: Error 1	33	TOHacks I: Error 2	1
Calculator II: Error 2	33	TOHacks I: Error 3	12
Calculator II: Error 3	33	TOHacks II: Error 1	1
Calculator II: Error 4	33	TOHacks II: Error 2	1
Calculator II: Error 5	33	Totems I: Error 1	1
Calculator II: Error 6	341	Totems I: Error 2	1
Chat II: Error 1	>500	Whiteboard I: Error 1	4
Game of Life I: Error 1	7	Whiteboard I: Error 2	9
Game of Life II: Error 1	8	Whiteboard II: Error 1	3
Game of Life II: Error 2	>500	Whiteboard II: Error 2	3
Slack Mockup I: Error 1	>500	Whiteboard II: Error 3	9
Slack Mockup I: Error 2	>500	Whiteboard II: Error 4	15
Slack Mockup I: Error 3	>500	Whiteboard II: Error 5	15

each high-priority error that was injected into an application how many test runs StackFul$_{\text{INTER}}$ required to reproduce this error, if at all, during its inter-process testing phase. Errors that were misclassified, e.g., the errors in Calculator I and Chat II, must have required *at least* 500 test runs before they are reached. Note that we assume that StackFul$_{\text{INTER}}$ would eventually reproduce a high-priority error in its inter-process testing phase, given an infinite test budget.

It can be observed that these errors fall into two categories: those errors which StackFul$_{\text{INTER}}$ can reproduce in a rather small number of test runs (i.e., 33 runs or less), and those for which it requires many more test runs. In Section 5.5.6, we hypothesise a likely explanation for this stark contrast.

5.5.5 Threats to Validity

We identify the threats to construct and external validity of our empirical evaluation.

Construct Validity

We have developed StackFul$_{\text{INTER}}$ to investigate the strengths and weaknesses of inter-process testing. It is worth noting that StackFul$_{\text{INTER}}$ lacks some optimisations that are generally included in automated testers such as state-of-the-art concolic testers. These optimisations would certainly have impacted the efficiency of both the intra-process and the inter-process testing phase with regards to the number of test runs required respectively to discover *any* server error, and to classify an error as a high-priority error. In the particular set of applications that we investigated, StackFul$_{\text{INTER}}$ discovered most injected server errors during the intra-process phase but was unable to reproduce some of these errors during the inter-process phase. We assert that additional optimisations to StackFul$_{\text{INTER}}$ would have prevented high-priority errors from being misclassified. We come back to this statement in Section 5.5.6.

External Validity

The main threat to the external validity of our evaluation is the limited number of programs on which we conducted it. We partially mitigated this threat by creating variants of the original programs in which we automatically and systematically inject faults to avoid possible bias.

5.5.6 Discussion of the Results

The results of Section 5.5.2 and 5.5.3 indicate that StackFul$_{\text{INTER}}$ is generally able to correctly categorise discovered errors. However, the results also highlight two related weaknesses in StackFul$_{\text{INTER}}$'s approach. First, if StackFul$_{\text{INTER}}$ requires more inter-process test runs than its test budget allows to reproduce a server error, the error is automatically classified as a low-priority error, even if the error is reachable in the given client-server configuration. All of the misclassified high-priority errors that were described in Section 5.5.2 and 5.5.4 suffer from this effect. Second, StackFul$_{\text{INTER}}$ can only assume that an error is of low priority in a particular client-server configuration if the error cannot be reproduced over the span of the inter-process testing phase. Therefore, StackFul$_{\text{INTER}}$ must always exhaust its entire test budget to classify a collection of low-priority errors. We discuss these weaknesses separately.

Misclassification of High-Priority Errors

The first weakness can be mitigated by employing more sophisticated search strategies during the inter-process testing phase. Investigating the number of required test runs as described in Section 5.5.4 more closely, it becomes apparent that the inter-process phase performs at its worst when attempting to reproduce server errors that are only reachable from the client by following a specific sequence of client-side events, while satisfying additional client-side branch conditions. This effect is particularly pronounced in the case of Slack Mockup, which is not only the largest benchmark but which also features the most intricate configuration of event handlers. StackFul$_{\text{INTER}}$ features two strategies for exploring event-driven applications: a simple brute-force strategy and another strategy that aims to maximise read-write conflicts between event handlers [72]. However, neither strategy is able to overcome this issue.

More efficient exploration strategies might help reduce the time StackFul$_{\text{INTER}}$ needs to reproduce an error, and therefore prevent StackFul$_{\text{INTER}}$ from misclassifying high-priority errors. Strategies that are better suited for testing highly event-driven applications might yield improved results. Several such exploration strategies for event-driven and message-driven applications have been described, such as dCute [114], which generates event sequences based on a partial ordering of the events, or Contest [7], which identifies subsuming event sequences. Another technique for prioritising event sequences or program paths that lead the client to communicate with the server is to initialise the test selector with program-specific *path prefixes* [101, 14]. These are programmer-defined sequences of events that are guaranteed to lead the tester to exercise a desired part of the application. All event sequences that are generated by the test selector must start with one of these prefixes. Another option consists of reducing the number of program paths that must be explored in order to cover the application, for example by applying *state merging*. This technique will be discussed in detail in Chapter 6.

Efficiency of Classifying Low-Priority Errors

The efficiency of StackFul$_{\text{INTER}}$'s inter-process testing phase would be improved if StackFul$_{\text{INTER}}$ could *prove* that a low-priority error is indeed unreachable in a given client-server configuration. This would prevent StackFul$_{\text{INTER}}$ from having to exhaust its entire inter-process test budget in order to classify low-priority errors. However, to prove that a server error is unreachable from a given client, it must be demonstrated that no program path in the entire web application starts from an entry point in the client (such as an event handler) and reaches the given error. In practice, even just the client side of the application is likely to have an infinite number of program paths, so exploring all of these to determine non-reachability is not possible.

5.6 Conclusion

We summarise the topics discussed in this chapter and conclude by evaluating the inter-process StackFul$_{\text{INTER}}$ tester on the criteria outlined in Section 2.4 for testing full-stack JavaScript web applications.

5.6.1 Summary

This chapter presented inter-process testing, which we defined as *testing the composition of all processes in a full-stack JavaScript web application as a whole, while observing their communication, and having their execution paths cross process boundaries.* We presented three concrete requirements for implementing inter-process concolic testing:

- Concolic testing must simultaneously take place on all processes of the application.
- The tester must construct a global path constraint that crosses the boundaries of the processes.
- The tester must be capable of modelling message payloads sent between the processes.

We implemented inter-process testing in StackFul$_{\text{INTER}}$, which is built on top of StackFul$_{\text{INTRA}}$. StackFul$_{\text{INTER}}$ can launch a variable number of instances of client and server processes, with the multiplicity of the processes configured by StackFul$_{\text{INTER}}$'s user at startup time. StackFul$_{\text{INTER}}$ uses Aran-Remote to deploy a single instance of a dynamic analysis across all of these processes. This in turn enables the tester to construct a global path constraint over the application across process boundaries. StackFul$_{\text{INTER}}$ furthermore detects registrations of Socket.IO message handlers and message sends, which allows it to inspect the message payloads transmitted between instrumented processes.

Inter-process testing realises an increase in precision when testing the program, as it maintains a global overview of the execution of the entire application. We motivated this increased precision by implementing a use case in StackFul$_{\text{INTER}}$ that allows for distinguishing between high-priority and low-priority server errors, based on whether the server error is reachable from a client in the full-stack JavaScript web application. We believe this distinction may prove useful for developers in that they may choose to prioritise patching high-priority errors over low-priority errors, as the first type of errors may be triggered on the server by any end user interacting with the application via a client. However, low-priority server side errors may still be triggered in theory, for example by having a malicious user bypass the client altogether. It may prove insightful to conduct a survey among developers of full-stack web applications into how they view the distinction between these errors, and how it conforms to their best practices.

In this use case, StackFul$_{\text{INTER}}$ is configured to test the application in a two-phase approach. In a first phase, StackFul$_{\text{INTER}}$ employs intra-process testing on just the server. In the second phase, StackFul$_{\text{INTER}}$ employs inter-process testing to test the entire application. StackFul$_{\text{INTER}}$ attempts to reach the previously discovered server errors from the application's clients. Any server error reproduced in the second phase is marked as a high-priority error, as this error can be triggered by an end-user interacting with the application. Errors that are only reached in the first phase, are marked as low-priority errors since they cannot be reproduced in this particular configuration of client and server processes.

We formalised this approach to inter-process testing in order to facilitate its replication, and we evaluated our use case with regards to its precision. Our evaluation on eight full-stack JavaScript web applications shows that StackFul$_{\text{INTER}}$ is capable of correctly classifying a majority of the high-priority server errors, without misclassifying any of the server's low-priority server errors.

5.6.2 Concluding Remarks

We conclude by evaluating the inter-process StackFul$_{\text{INTER}}$ concolic tester on the seven criteria for performing automated testing on full-stack JavaScript web applications. Table 5.4 illustrates which of these criteria have been satisfied.

Table 5.4: The inter-process StackFul$_{\text{INTER}}$ tester evaluated on the list of criteria identified in Section 2.4.

Criterion		Description	Satisfied?
①	A	Capable of testing sequential JavaScript code	✓
	B	Models inputs from the DOM	✓
②	A	Allows for dynamic (de)registration of event handlers	✓
	B	Explores event space	±
③		Capable of finding composition-specific faults	✓
④	A	Whole-program monitoring	✓
	B	Observes communication between distributed processes	✓

Criteria 1-2

These criteria concern the testing of single-process, event-driven JavaScript applications which have access to a DOM (e.g., websites). Recall that StackFul$_{\text{INTRA}}$ satisfies three of these criteria. It only partially satisfies criterion 2.B, as the efficiency with which the tester is capable of exploring the event space strongly depends on the search strategy that it employs.

StackFul$_{\text{INTER}}$ is built directly on top of StackFul$_{\text{INTRA}}$ and neither extends nor changes the characteristics that made StackFul$_{\text{INTRA}}$ capable of satisfying these four criteria. StackFul$_{\text{INTER}}$ hence inherits StackFul$_{\text{INTRA}}$'s classicication for these criteria.

Criterion 3

Criterion 3 relates to whether a tester is capable of finding bugs that only manifest themselves in specific compositions of processes, such as errors that only appear when a fixed number of clients have connected to the server.

StackFul$_{\text{INTER}}$ can launch a variable number of instances of client and server processes in a full-stack JavaScript web application. It does not limit the number or type of processes that are launched, nor does it restrict the composition of these processes. Rather, when StackFul$_{\text{INTER}}$ starts testing an application, it follows a user configuration that specifies which processes should be launched, and in what order. This composition is fixed at startup time and cannot be changed dynamically.

Given a user configuration for a composition of processes in which a bug may manifest itself, StackFul$_{\text{INTER}}$ can follow this configuration and detect the bug. We conclude that StackFul$_{\text{INTER}}$ is capable of finding composition-specific faults, and hence mark criterion 3 as satisfied.

Criterion 4

Criterion 4 concerns the multi-process nature of a full-stack JavaScript web application. It consists of two subcriteria.

Subcriterion 4.A requires the tester to monitor the execution of all processes simultaneously. StackFul$_{\text{INTER}}$ employs a combination of Aran and Aran-Remote to deploy a single instance of a dynamic analysis over multiple processes. Like StackFul$_{\text{INTRA}}$, this analysis enables shadow execution of each process, which in turn allows for performing symbolic execution and collecting symbolic constraints alongside the concrete execution of the process. Because the shadow execution is performed across all processes, StackFul$_{\text{INTER}}$ can construct a global path constraint that crosses process boundaries for the entire application. The first subcriterion is hence satisfied.

Subcriterion 4.B requires the tester to observe the communication between the distributed processes. StackFul$_{\text{INTER}}$ observes and intervenes in the communication between processes that takes place over Socket.IO websockets. When the application's code registers a message handler, StackFul$_{\text{INTER}}$ intervenes and registers a replacement callback instead so that it can observe and inspect the messages received by a process. It likewise intervenes in the sending of Socket.IO messages. This enables the tester to model the symbolic expressions corresponding to the concrete values in the message's payload precisely. We therefore mark subcriterion 4.B as satisfied.

Overall Conclusion

StackFul_{Inter} fully satisfies six of the seven criteria for automated testing of full-stack JavaScript web applications. Criterion 2.B still remains only partially satisfied, as the extent to which the tester is capable of efficiently exploring the application's event space strongly depends on the search strategy that it employs. A simple brute-force exploration of the event space is insufficient for non-trivial applications, as the number of execution paths through an application grows exponentially in function of the number of event handlers that are registered and their complexity. Chapter 6 describes an approach to state merging which addresses this exponential blowup and which will allow our tester to fully satisfy the last remaining criterion.

6 State Merging for Event-driven Programs

The previous chapter introduced StackFul$_{\text{INTER}}$, an inter-process concolic tester for full-stack JavaScript web applications. As described in Section 5.6, this tester satisfies six of the seven criteria for testing these types of applications. The last criterion that remains unsatisfied concerns the efficient exploration of the application's search space. Satisfying this criterion is complicated by the nature of full-stack JavaScript web applications, as a tester for these applications must not explore just one process, but several processes simultaneously, as well as their interplay.

A traditional concolic tester explores each unique path through the program separately, requiring a separate test run to do so. For a given path constraint consisting of n branch constraints, the number of variations that this path constraint gives rise to is generally exponential in n. A longer path constraint, spanning several processes, therefore results in an exponential increase in the number of additional program paths that must be explored by the tester. It is therefore important to consider how StackFul$_{\text{INTER}}$ can manage these large search spaces.

This chapter describes how the search space of full-stack JavaScript web applications can be reduced by merging program states that are *sufficiently similar* together, without sacrificing precision of the symbolic constraints. We present a novel approach for transposing this known *state merging* technique [99] to the domain of *offline* symbolic execution (i.e., concolic testing) of event-driven applications. By rendering state merging possible for the concolic testing of event-driven applications, we can also apply it to full-stack JavaScript web applications. We outline how StackFul$_{\text{INTER}}$ is extended with an implementation of this novel technique, and we name the version of StackFul$_{\text{INTER}}$ that incorporates this form of state merging *StackFul*[1].

[1] `https://github.com/softwarelanguageslab/StackFul`

Section 6.1 gives an overview of state merging for *online* symbolic execution of sequential applications that are not event-driven. Section 6.2 continues by transposing state merging to the setting of concolic testing. Section 6.3 takes this a step further by describing how state merging can be applied by concolic testers to event-driven applications. In Section 6.4, we give an overview of the implementation of this approach in StackFul. Section 6.5 evaluates the impact of state merging on the same suite of input programs that was used to evaluate inter-process testing (cf. Section 5.5). We conclude in Section 6.6 by discussing the effectiveness of state merging on the testing of full-stack JavaScript web applications.

6.1 Overview of State Merging

Section 3.5 listed several generic techniques for rendering concolic testing more effective when testing applications with a large search space. These techniques were categorised either as using heuristics to steer the tester towards desired parts of the application (Section 3.5.1), as mitigating the path explosion problem (Section 3.5.2), or as combining various approaches to automated testing (Section 3.5.3). Of these solutions, only the path explosion mitigation techniques aim to prevent or alleviate an exponential blow-up of program paths.

Path Explosion Mitigation Techniques

The idea behind these techniques is to make it possible for a tester to fully explore a program without having to test each unique path through this program.

One example of a path explosion mitigation technique is compositional symbolic execution, which is based on the idea that once all paths through a particular code fragment (such as a function, loop, or block) have been explored, a symbolic summary for this fragment can be generated. This summary details which symbolic outputs (e.g., return values, assignments to global variables, etc.) can be expected for the symbolic inputs (e.g., arguments) to the fragment. Whenever execution again passes through this snippet, the tester pauses symbolic execution and appends the previously generated summary to the current path constraint. Thus, rather than spawning new program paths wile executing the code, the tester can reuse the summary, thereby reducing path explosion.

```
if (randomInt() === 0) {  }
else {  }

if (randomInt() === 1) {  }
else {  }
// program ends
```

Listing 6.1: A simple program that gives rise to four program paths.

State merging is another example of path explosion mitigation. Broadly speaking, *states* correspond to the nodes in the symbolic execution trees presented throughout this dissertation, although we revisit this definition of states in Section 6.1.3. States are created alongside paths and encode a snapshot of the current point in the symbolic execution. States include path constraints, so an exponential increase in the number of paths (*"path explosion"*) corresponds to a likewise exponential increase in the number of states (*"state explosion"*) and vice versa [66]. The two terms are hence interchangeable. State merging alleviates the state explosion problem by merging program states that are sufficiently *similar* together. Instead of having states in the symbolic execution tree split whenever a branch condition is encountered, two or more states can be merged together, thereby also halving the total number of states that descend from these two states.

To illustrate this, consider the simple example program in Listing 6.1 which gives rise to four program paths, i.e., four `END` states, when executed symbolically. Its corresponing symbolic execution tree is shown in Figure 6.1. The figure includes the program's four `END` states, along with their path constraints.

Figure 6.2 depicts the result of merging the two states that feature the constraint $i_1 = 1$ together. Note that after applying the state merge operation, the symbolic execution tree has transformed into a symbolic execution Directed Acyclic Graph. (DAG). By performing one merge operation, the number of `END` states is halved. Furthermore, although the number of states has been reduced, no information on the path constraints through the DAG has been lost: the path constraints for the two `END` states now feature a logical disjunction that includes both branches of the root $i_0 = 0$ state. These two path constraints hence still precisely encode the information that was contained in the four path constraints for the `END` nodes of the unmerged symbolic execution tree. Note that, although the number of `END` states has been reduced as a result of the merge operation, the complexity of their path constraints has increased.

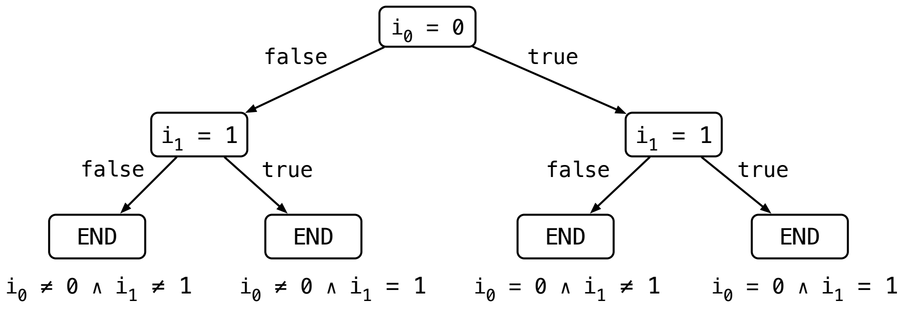


Figure 6.1: The symbolic execution tree for the code in Listing 6.1.

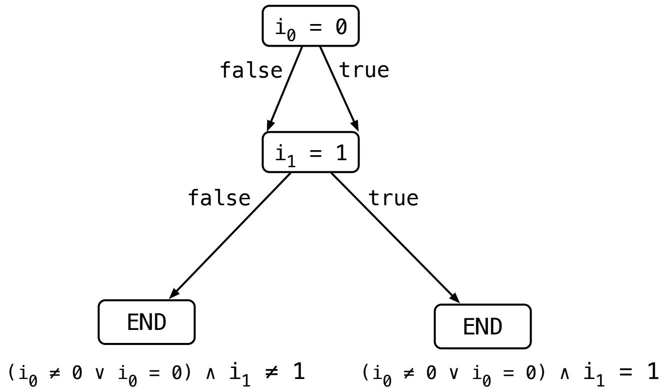

$i_0 = 0$
false
true
$i_1 = 1$
false
true
END
END
$(i_0 \neq 0 \vee i_0 = 0) \wedge i_1 \neq 1$
$(i_0 \neq 0 \vee i_0 = 0) \wedge i_1 = 1$

Figure 6.2: After merging two states together, the number of `END` states is halved.

An essential difference between compositional symbolic execution and state merging lies in their granularity. Compositional symbolic execution summarises the execution of whole code fragments (e.g., functions, loop, blocks). In order to create these summaries, the tester has to explore all paths through the targeted scope, even if these paths are unlikely to occur in practice. Compositional execution furthermore does not prevent state explosion from occurring within the targeted scope.

State merging on the other hand can be much more fine-grained as it is capable of merging different states at an individual level together. Its granularity is dependent on the similarity metrics employed by the technique: if the tester employs a similarity metric that finds a large set of states to be similar, many of these states will be merged together. A more narrow similarity metric will result in the merging of fewer states.

6.1.1 Revisiting the State Explosion Problem

As an example of the utility of state merging, consider Listing 6.2. This code snippet consists of five sequential, non-nested `if` statements. The predicates of the first four `if` statements are all independent from each other. On the other hand, the last `if` statement depends entirely on the first four `if` statements: its predicate depends on the values of the variables `a`, `b`, `c`, and `d`. Execution can hence only follow its `then` branch if all four previous `then` branches were followed.

```
let a, b, c, d;
if (randomInt() === 0) {
  a = 1;
} else {
  a = 0;
}
if (randomInt() === 1) {
  b = 1;
} else {
  b = 0;
}
if (randomInt() === 2) {
  c = 1;
} else {
  c = 0;
}
if (randomInt() === 3) {
  d = 1;
} else {
  d = 0;
}
if (a + b + c + d === 4) {
  // program ends, via then branch
} else {
  // program ends, via else branch
}
```

Listing 6.2: A small code snippet that results in an exponential increase in the number of program paths.

The number of unique program paths that can be followed through this snippet, corresponding to the set of unique values that variables `a`, `b`, `c`, and `d` can assume, equals $2^4 = 16$ paths. Figure 6.3 depicts the symbolic execution tree for Listing 6.2, with unreachable nodes coloured black. In this figure, the left child edge of a node always corresponds to the `else` branch of the node's associated `if` statement, and the right edge corresponds to its `then` branch. In this tree, the symbolic predicate for the first `if` statement occurs only once, the symbolic predicate for the second `if` statement occurs twice, the third predicate is listed four times, the fourth predicate eight times, and the last node, with predicate `a & b & c & d`, is repeated sixteen times. For clarity, we also include the leaf nodes `END T` and `END F` which indicate that program execution terminated either on line 23 or at 25. As traditional concolic testing explores each path separately, a traditional concolic tester hence requires sixteen test runs to explore all feasible paths through the snippet.

For the leftmost fifteen occurrences of the `a + b + c + d = 4` node, the child node that corresponds to their `then` branch (coloured black in the figure) is unreachable, as these occurrences correspond to program paths where at least one of the variables `a` through `d` is assigned `0`. For these fifteen nodes, execution can only reach their `else` branch. This situation is reversed for the rightmost occurrence of the `a + b + c + d = 4` node, where execution is forced along the `then` branch and the `else` branch is unreachable.

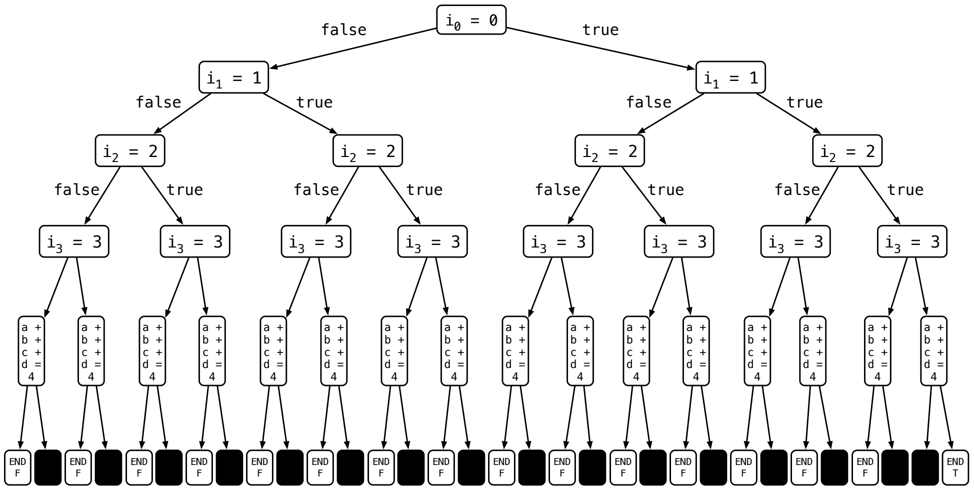


Figure 6.3: The symbolic execution tree for the code snippet in Listing 6.2.

The path explosion arises from the fact that execution splits immediately after reaching the first `if` statement (i.e., the root node with symbolic predicate $i_0 = 0$) into two new paths: one that is created by following the `then` branch and another that is created by following the `else` branch. Not only do these two paths stay separate for the remainder of the program's execution, but they each subsequently split several times into sub-paths upon encountering each next `if` statement. Had these two initial paths been rejoined after their split at the $i_0 = 0$ node but before their next split at the $i_1 = 1$ node, the symbolic execution tree would have remained fairly small, instead of exploding into sixteen different `END` leaf nodes and an additional sixteen unreachable nodes.

6.1.2 Alleviating State Explosion through State Merging

Every node in the symbolic execution tree depicted in Figure 6.3 represents a specific *state* that was observed while executing the program. For example, an occurrence of a node $i_2 = 2$ in the tree corresponds to a particular state that was created when execution reached the third `if` statement along a particular program path.

Recall that the path explosion problem can be cast as a state explosion problem, which in turn can be solved by merging *similar* states together, i.e., states that correspond to the same `if` statement in the tree, rather than duplicating a state for each program path that reaches this statement. Figure 6.4 depicts what the symbolic execution DAG for Listing 6.2 would be if all states that correspond to the same `if` statement in the code were merged together.

Each program path gives rise to a unique combination of values assigned to the variables `a` through `d`. The difference in values for these variables is significant when the tester explores both branches of the last `if` statement on line 22 of Listing 6.2. Of the sixteen program paths that encounter this `if` statement, only one path visits the `then` branch, since only one path sets the value of all four variables to `1`. When joining states together into one merged state, the tester must hence be capable of precisely modelling the different values that each of the four variables assumes depending on whether the `then` branch or `else` branch was taken when assigning the variable.

State merging therefore employs symbolic *if-then-else* (ITE) expressions to model symbolic values that depend on some condition. For example, the value of the variable `a` can be represented as $\mathit{ITE}(i_0 = 0, 1, 0)$: this value evaluates to 1 if the predicate $i_0 = 0$ is true, and evaluates to 0 otherwise. Using these ITE expressions, the tester can represent the predicate of the last `if` statement as $\mathit{ITE}(i_0 = 0, 1, 0) + \mathit{ITE}(i_1 = 1, 1, 0) + \mathit{ITE}(i_2 = 2, 1, 0) + \mathit{ITE}(i_3 = 3, 1, 0) = 4$.

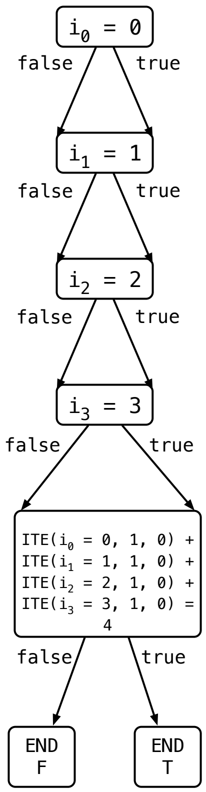

i0 = 0
false
true
i1 = 1
false
true
i2 = 2
false
true
i3 = 3
false
true
ITE(i0 = 0, 1, 0) + ITE(i1 = 1, 1, 0) + ITE(i2 = 2, 1, 0) + ITE(i3 = 3, 1, 0) = 4
false
true
END F
END T

Figure 6.4: The symbolic execution DAG for Listing 6.2 with state merging.

Note that these ITE expressions may be computationally expensive to solve for an SMT solver, especially when they are included into path constraints that already feature long chains of logical disjunctions (as depicted in Figure 6.2). The increased complexity of these path constraints increases the burden on the SMT solver that must compute appropriate values for the symbolic input variables that are featured in these paths. In effect, state merging reduces the time that the tester spends exploring duplicate states, but increases the solving time of the SMT solver. However, for concolic testing, state merging is still beneficial in itself as the tester incurs an additional overhead in executing the program. By ensuring that no duplicate states are executed, this execution overhead is reduced.

6.1.3 A Formal Definition of State Merging

State merging can be defined formally in terms of state triples $\langle P, PC, \sigma\rangle$, where PC represents the path constraint that leads to this state in the tree, σ the symbolic store mapping program variables to symbolic values, and P the *program point*. The program point encodes the current point in the program, e.g., a specific `if` statement, that was being executed when the state was created.

The State Merge Operation

The state merge operation $\sim$ is then defined on two states $\langle P, PC_1, \sigma_1\rangle \sim \langle P, PC_2, \sigma_2\rangle = \langle P, PC_1 \vee PC_2, \sigma_m\rangle$, where σ_m is defined as $\forall v \in \sigma_1 : \sigma_m[v] = ITE(PC_1, \sigma_1[v], \sigma_2[v])$.

In effect, this merged triple defines a state which can be reached by following either path constraint PC_1 or PC_2. Its symbolic store σ_m maps every program variable to an ITE expression. If the path constraint PC_1 of the first state evaluates to `true`, this ITE expression itself evaluates to the symbolic value observed by the first state and kept in its symbolic store σ_1. Note that this definition assumes that both symbolic stores include the exact same variables, though their values may be different.

If this merged state is merged again with another state $\langle P, PC_3, \sigma_3\rangle$, we apply the same merge operation. The merge operation then produces the state $\langle P, PC_3 \vee PC_1 \vee PC_2, \sigma_{m'}\rangle$, where $\sigma_{m'}$ is defined as $\forall v \in \sigma_3 : \sigma_{m'}[v] = ITE(PC_3, \sigma_3[v], \sigma_m[v])$, which is equivalent to $\forall v \in \sigma_3 : \sigma_{m'}[v] = ITE(PC_3, \sigma_3[v], ITE(PC_1, \sigma_1[v], \sigma_2[v]))$ Note that the three path constraints PC_1, PC_2, and PC_3 do not necessarily exhaustively cover all possible execution paths through a program. The negation of PC_3 and PC_1 therefore does not generally imply that PC_2 is true, and hence that any program variable `x` should receive the value $\sigma_2[\texttt{x}]$ along a path that differs from PC_3 and PC_1. However, *with respect to the merged state's path constraint* $PC_3 \vee PC_1 \vee PC_2$, the symbolic store $\sigma_{m'}$ represents all three possibilities. As new states featuring the same program point are produced while testing the program, these states may also be joined together with this merged state. Both the merged state's path constraint and its store will then cover all feasible program paths more and more exhaustively.

Defining the Program Point

The merge operation is only applied to two states that share an identical program point P. This program point is defined only opaquely, but it is taken to refer to a specific point in the program's execution. In sequential programs without recursion or loops, such as the code snippet in Listing 6.2, it could be defined as just the code label (e.g., the line number) that was reached by the tester when the state was created. This makes it reasonable to assume that both stores include the same set of variables, as this definition of program points ensures that states that are merged together were created in the same lexical environment. We revisit the definition for program points in Sections 6.2.1 and 6.3.2.

Example of a State Merge Operation

As an example, consider the formal representation of both occurrences of the state $i_1 = 1$. Defining program points as line numbers, the leftmost (as depicted in Figure 6.3) of these two states can be represented by the triple $\langle L_7, i_0 \neq 0, \sigma_1 \rangle$, where σ_1 is defined as a singleton map of the variable `a` to the symbolic value 0. The rightmost of these states is represented as the triple $\langle L_7, i_0 = 0, \sigma_2 \rangle$, with σ_2 mapping the variable `a` to the symbolic value 1. The result of the merge operation would then be defined as the triple $\langle L_7, i_0 = 0 \vee i_0 \neq 0, \sigma_m \rangle$, with σ_m mapping the variable `a` to the symbolic value $\mathit{ITE}(i_0 = 0, 1, 0)$.

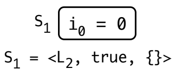
6.1.4 State Merging for Online Symbolic Execution

The formal definition of state merging can be used by any type of symbolic executor to merge two given states together. However, this definition is insufficient for understanding how state merging can be incorporated into the symbolic execution procedure at large, or how it affects the exploration of a program. We therefore give an overview of how state merging can be integrated with symbolic execution. As an introduction, we first detail how state merging can be incorporated into *online* symbolic execution of *sequential* applications, before describing how the state merging algorithm can be transposed to the setting of concolic testing (Section 6.2), and afterwards concolic testing of event-driven applications (Section 6.3).

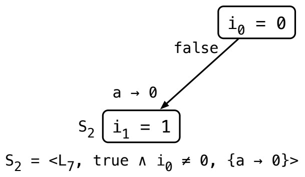
Online symbolic executors are symbolic executors that can fork their execution when reaching a branching point [13, 27]. These stand in contrast to *offline* symbolic executors, like concolic testers, which test an application by exploring one program path at a time. Online symbolic executors can select which state in the symbolic execution DAG to expand. Whenever the online executor has expanded a previously unexplored state in the symbolic execution DAG, it can compare its program point with that of all other explored states in the DAG. If it finds a previous state with an identical program point, the symbolic executor may opt to merge the two together before continuing its exploration of the remainder of the symbolic execution DAG.

Figure 6.5 illustrates an example of how an online symbolic executor can incorporate state merging while testing the code from Listing 6.2. As execution in this code snippet does not jump back to an earlier point in the program, via e.g., recursion or iteration, it is sufficient to represent program points as just the line number of the code statement being executed. For each new state that is added to the symbolic DAG, we list the triple representing that state.

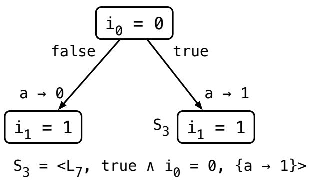
Apart from a `true` and `false` label, we also annotate the edges in the DAG with the assignments to program variables that were observed to have taken place when following the branch. We also annotate merged states where program variables have received a symbolic ITE expression as a result of the merge. An example of such a state annotation can be seen in e.g., Figure 6.5d, where the state annotation (highlighted in bold), indicates that the value of `a` is replaced by an ITE expression. The symbolic store for any state S can hence be constructed via the annotations along both the edges and the states on the path to S.



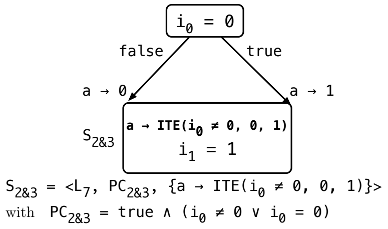
(a) After evaluating the branch condition on line 2, the root state is created.



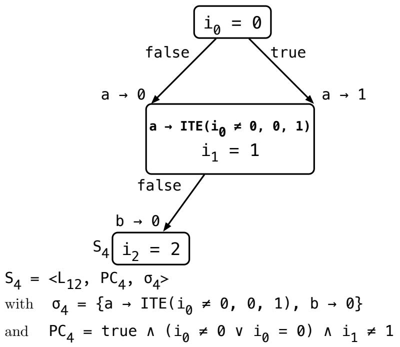
(b) The symbolic executor expands the **else** branch of the previous condition and adds a state corresponding to the branch condition on line 7.



(c) The symbolic executor expands the **then** branch of the previous condition and adds a state corresponding to the branch condition on line 7.


i0 = 0
false
true
a → 0
a → 1
S2&3
a → ITE(i0 ≠ 0, 0, 1)
i1 = 1
S2&3 = <L7, PC2&3, {a → ITE(i0 ≠ 0, 0, 1)}>
with PC2&3 = true ∧ (i0 ≠ 0 ∨ i0 = 0)

(d) Having expanded both child states, the symbolic executor immediately merges them.



(e) The symbolic executor continues testing the program by executing the **if** statement on line 12.

Figure 6.5: Online symbolic execution of the code in Listing 6.2, using state merging to prune states with duplicate program points.

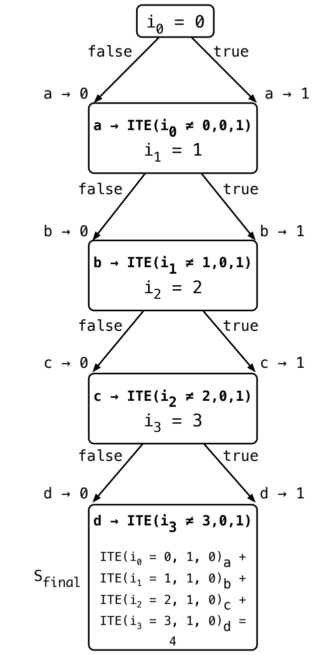
The symbolic executor first adds a state S_1 for the first `if` statement of the code snippet (Figure 6.5a). As the root state in the symbolic DAG, the state features a path constraint with the default value `true` and an empty symbolic store.

It continues along the `else` branch of this `if` statement, injects the mapping from variable `a` to the symbolic value 0 into the symbolic store when observing the assignment to this variable, and then adds a new state S_2 $\langle L_7, \texttt{true} \wedge i_0 \neq 0, \{a \rightarrow 0\}\rangle$ to the symbolic DAG upon reaching the next `if` statement (Figure 6.5b). Before continuing to the child states of this `if` statement, the executor returns to the root state and explores the `then` branch of the first `if` statement. As before, the tester adds the mapping `a` $\rightarrow$ 1 to its store and adds a duplicate state S_3 $\langle L_7, \texttt{true} \wedge i_0 = 0, \{a \rightarrow 1\}\rangle$ as the `then` child to the root (Figure 6.5c).

Since both states share the same program point, the symbolic executor merges them together into state $S_{1\&2}$. The path constraint of the merged state features the disjunction of both original states: $(\texttt{true} \wedge i_0 \neq 0) \vee (\texttt{true} \wedge i_0 = 0) \equiv \texttt{true} \wedge (i_0 \neq 0 \vee i_0 = 0)$. The variable `a` in the merged symbolic store is mapped to the symbolic value $\mathit{ITE}(i_0 \neq 0, 0, 1)$.

Upon completing the merge operation, the symbolic executor resumes normal symbolic execution of the program by exploring the `else` branch of the merged state. The constraint $i_1 \neq 1$ is added to the path constraint of the merged state, while the mapping `b` $\rightarrow 0$ is added to the symbolic store alongside the mapping for `a`.

Figure 6.6 depicts the final symbolic execution DAG that is produced once symbolic execution of the whole program finishes. The bottom state corresponds to the merged state for the `if` statement on line 22. The four variables that were used in this `if` statement have been replaced by a corresponding ITE expression. For clarity, we subscript ITE expressions that were produced as a result of reading a variable with the name of the variable that was inlined.



$$
\begin{aligned}
S_{final} &= \langle L_{22}, PC_{final}, \sigma_{final}\rangle \\
\text{with}\quad \sigma_{final} &= \{a \to ITE(i_0 \neq 0, 0, 1), \\
&\quad b \to ITE(i_1 \neq 1, 0, 1), \\
&\quad c \to ITE(i_2 \neq 1, 0, 1), \\
&\quad d \to ITE(i_3 \neq 1, 0, 1)\} \\
\text{and}\quad PC_{final} &= true \\
&\quad \wedge (i_0 \neq 0 \vee i_0 = 0) \\
&\quad \wedge (i_1 \neq 1 \vee i_1 = 1) \\
&\quad \wedge (i_2 \neq 2 \vee i_2 = 2) \\
&\quad \wedge (i_3 \neq 3 \vee i_3 = 3)
\end{aligned}
$$

Figure 6.6: The complete symbolic execution DAG, produced after symbolic execution of the program has finished.

Practical Considerations

Note that it may not always be beneficial to merge states together. The introduction of symbolic ITE expressions increases the burden on the SMT solver, as it becomes more computationally intensive to solve queries containing these expressions [55, 113], especially since these ITE expressions may be nested. If merging introduces too many ITE expressions, the computational cost of solving queries may outweigh the cost of exploring duplicated program paths. Section 3.5.2 describes several techniques for finding a balance between the computational overhead introduced by state merging and the effort saved by the tester in exploring redundant paths. Of note is that it may be possible to heuristically predict the computational cost of replacing “concrete” symbolic values, such as 1 and 2, with more abstract symbolic values such as $ITE(i_0 = 0, 1, 2)$ [66]. Merging then only takes place when the heuristic determines that the cost of replacing these values with an ITE expression is outweighed by the reduction in available program paths.

6.2 State Merging for Concolic Testing

The previous section gave an overview of how state merging is traditionally realised for online symbolic execution, where multiple program paths are explored simultaneously and where execution can be forked at every state. In this section, we give an overview of how state merging can be realised in the setting of an offline executor, such as a concolic tester, which is restricted to exploring a program one path per time.

6.2.1 Complications for Applying State Merging

The previous overview of state merging for online symbolic execution was aided by two factors:

- The program point could be represented as just a code label (e.g., the line number). In real-world applications, a code label does not suffice to precisely define the execution point of the state.
- The symbolic executor was able to backtrack to any previous state in the symbolic DAG, which in turn allowed for interleaving state merging and symbolic execution.

Defining Program Points

The code in Listing 6.3 demonstrates why the program point of a state cannot be modelled as just the code label of the corresponding branching point. The `if` statement on line 3 is reached via two different paths, namely the paths from functions `f` and `g`. Two different states would therefore be created in the symbolic execution tree to represent this branching point. If program points were to be defined as just the line number of the corresponding branching point, both states would be merged together. However, execution after this statement proceeds differently along both paths immediately upon returning from the `print` function. As the `if` statements encountered along both branches differ from each other, the structure of the subtrees of both states is also different. It may therefore be better to keep them separate and extend our definition of program points to reflect the different nature of both states.

We solve this problem by including the function stack in the program point. The program point for the state on line 3 that was reached via the function `f` is represented as the pair $\langle L_3, [\texttt{f}]\rangle$, whereas the program point for the same line reached via the function `g` is represented as $\langle L_3, [\texttt{g}]\rangle$. Since these program points are not identical, the symbolic executor does not consider the corresponding states as candidates for merging.

Note that including function stacks in program points is still insufficient in the general case. Section 6.3.2 discusses how program points should be redefined again when testing event-driven code, and Section 6.4.2 presents a suitable definition of program points for code featuring program loops.

```
1  let x, y;
2  function print(s) {
3    if (...) {
4      console.log(s);
5    }
6  }
7  function f() {
8    ... // elided
9    print("f");
10   if (x + y > 42) { ... } else { ... }
11 }
12 function g() {
13   ... // elided
14   print("g");
15   if (randomInt() === 0) { ...} else { ... }
16 }
```

Listing 6.3: An example code snippet where it does not suffice to define a program point as just a code label.

Interweaving Symbolic Execution and State Merging

Recall from the demonstration of state merging in the online symbolic execution of Listing 6.2 that the symbolic executor, after having explored the `else` branch of the root state (Figure 6.5b), backtracked to the root state to first explore the `then` branch of the root (Figure 6.5c), and then immediately joined both child states together (Figure 6.5d) before continuing the execution by exploring the `else` branch of the merged state (Figure 6.5e).

The symbolic executor was hence able to choose which states to explore in function of which states can be merged. This feature allows the executor to merge suitable child states as soon as they emerge, rather than first exploring an entire subtree for each state and having to merge these subtrees together. Backtracking hence enables the symbolic executor to avoid having to merge entire subtrees. However, this backtracking is not possible for offline symbolic executors, such as concolic testers, as we will discuss below.

6.2.2 Mitigating the Path Determinacy Problem

In the context of offline symbolic execution, i.e., concolic testing, it is not possible to strictly interleave exploration of the symbolic execution DAG and merging of duplicated states. Conceptually, the problem is that the program is executed both concretely and symbolically, but symbolic execution is forced to follow alongside the concrete execution. Unlike online symbolic executors, concolic testers cannot halt exploration of a particular path, backtrack to an earlier state, and resume exploration from its alternative branch.

Recall from Section 4.3 that before each test run the test selector specifies a particular path to be explored in the run. The test selector then employs an SMT solver to compute appropriate concrete values for the program inputs appearing in the constraints along that path. When the test run commences, the concolic tester is forced to proceed along this exact path, without opportunities to deviate from it or to temporarily explore a different branch. We name this the *path determinacy problem*, as the path, and hence the states that it will encounter along the path, is selected before a test run is started and must be followed until completion.

In practical terms, this means that the tester is limited to merging the states that it will find along the preselected path. To partially mitigate this problem, our state merging algorithm mimics that of merging in online symbolic execution as closely as possible, by incrementally adding a new path to the DAG in segments, rather than adding the whole path in one operation as described in Section 2.2.3. Whenever a partial path has been added to the DAG, the algorithm attempts to find a duplicate state.

The test selector still determines which path must be explored in the next test run, but the test executor attempts to perform a merge every time a new state is created, i.e., whenever it observes a branching condition. The `mergeState` function, which gives a pseudocode overview of the merging algorithm, is defined in Algorithm 3. Conceptually, the test executor invokes the `mergeState` function whenever a merge should be attempted.

An Algorithmic Description of State Merging for Concolic Testing

`mergeState` receives as input the root state of the symbolic execution DAG, which is of type State. The definition of a State structure is provided by Algorithm 2, which follows the previously described tuple definition of a state. A State includes a ProgramPoint, defined in Algorithm 1, which follows the extended definition of a program point described in Section 6.2.1.

ALGORITHM 1: Definition of a ProgramPoint struct.

```
struct ProgramPoint
    $lineNumber$ : number
    $functionState$ : A list of function labels.
end struct
```

ALGORITHM 2: Definition of a State struct.

```
struct State
    $programPoint$ : A ProgramPoint
    $PC$ : The path constraint.
    $store$ : A mapping of program variables to symbolic expressions.
end struct
```

`mergeState` receives as a second input the partial path that has been explored by the test executor in the current test run. `mergeState` first adds this partial path to the symbolic execution DAG (line 2) by calling the auxiliary function `addPathToDAG`. It then attempts to find a state previously inserted into the execution DAG with a program point that is identical to that of the last state of the partial path (line 3). If no duplicate program point can be found (line 4), the merge operation is aborted. If a suitable candidate state. named duplicateState, exists, the merge operation continues by constructing the merged symbolic store by calling the auxiliary function mergeStores (line 5).

mergeStores creates a new store by traversing all variables in both stores and comparing their values. If their values are different, the function inserts into the new store an ITE expression which uses as predicate the path constraint of lastState, and which has as then and else values the values from respectively the lastState's store and duplicate state's store. If the values are the same, the function simply inserts this value into the new store. We elide the definition of mergeStores, but it aligns with the definition of the state merge operation presented in Section 6.1.3: $\forall v \in \sigma_1 : \sigma_m[v] = \mathit{ITE}(PC_1, \sigma_1[v], \sigma_2[v])$, where σ_m is the store produced by the mergeStores function, and σ_1 and σ_2 are the symbolic stores of respectively lastState and duplicateState.

The merged state itself is then constructed (line 7) by creating a new State that consists of the unchanged *programPoint* of the duplicate state, the disjuncted path constraints of the two original states (created by calling the auxiliary function disjunct on line 6), and the merged store.

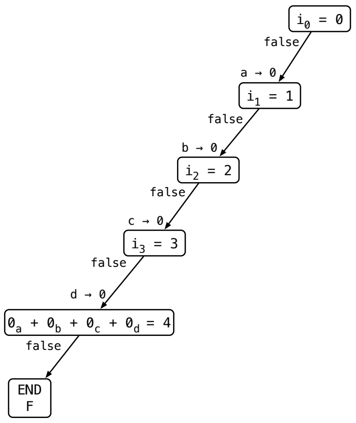
As a consequence of the path determinacy problem, duplicateState likely had a subtree. The symbolic stores of all states in this subtree have to be updated to reflect the new values of the program variables in mergedStore. Note that lastState does not have a subtree, as it was the most recent node to be added to the symbolic execution tree. The symbolic stores of all the states in the duplicate's subtree are updated via the auxiliary function applyStoreToChildren (line 8), returning a new subtree. This subtree is then added to mergedState (line 9).

Lastly, the original duplicate state and last state of the newly added path are replaced by having their parents (lines 10-11) replace these children with the newly merged state (lines 12-13).

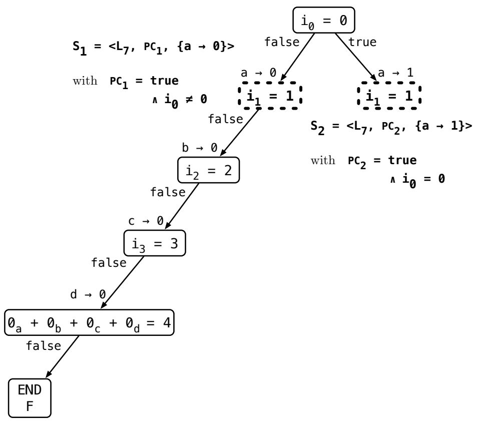
Figure 6.7 illustrates how the state merging algorithm for concolic testing is applied to the code depicted in Listing 6.2 over the course of three test runs. As before, when a variable is read, the inlined ITE expression is subscripted with the name of the variable.

ALGORITHM 3: A pseudo-code implementation of the state merging procedure for concolic testing.

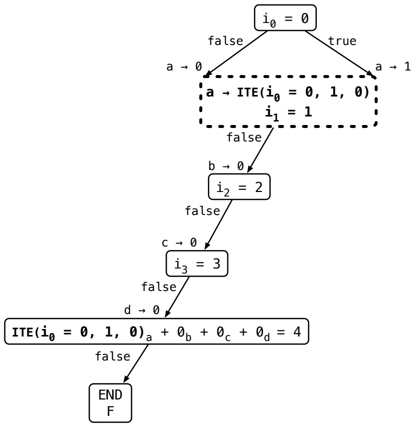
```
input : The root State that forms the root of the current symbolic
        execution tree.
input : The partial path constraint path that has been observed by the
        test executor.
output: A boolean indicating whether the merge operation was
        successful.
1  Function mergeState(root, partialPath)
2      lastState ← addPathToDAG(root, partialPath)
3      duplicateState ← getDuplicate(root, lastState.ProgramPoint)
4      if duplicateState ≠ null then
5          mergedStore ← mergeStores (lastState.PC, lastState.store,
             duplicateState.store)
6          disjunctedPC ← disjunct(duplicateState.PC, lastState.PC)
7          mergedState ← new State(lastState.ProgramPoint, disjunctedPC,
             mergedStore)
8          newSubtree ← applyStoreToChildren(mergedStore,
             duplicateState)
9          setSubtree(mergedState, newSubtree)
10         lastStateParent ← getParent(lastState)
11         duplicateStateParent ← getParent(duplicateState)
12         replaceChildWith(lastStateParent, lastState, mergedState)
13         replaceChildWith(duplicateStateParent, duplicateState,
             mergedState)
14         return true
15     else
16         return false
17     end
18 end
```



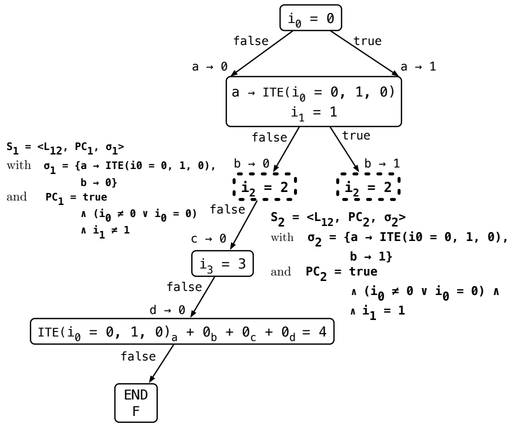
(a) Test run 1: The symbolic execution tree after adding the first program path.



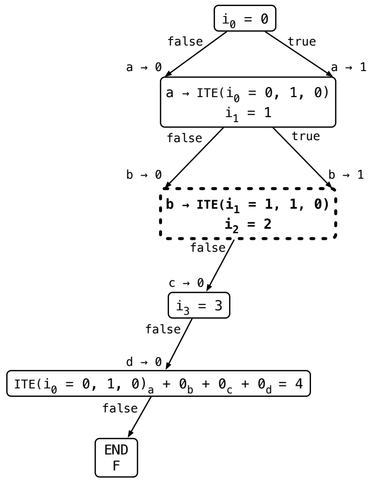
(b) Test run 2: The symbolic execution tree after adding part of the second program path, with the candidate states for drawn with a dotted line.



(c) Test run 2: The symbolic execution DAG after merging the candidate states from the previous step together.



(d) Test run 3: The symbolic execution DAG after adding part of the third program path.



(e) Test run 3: The symbolic execution DAG after merging the candidate states from the previous step together.

Figure 6.7: Concolic testing of the code in Listing 6.2, using state merging to prune similar states.

6.3 State Merging for Event-driven Applications

The previous section described how state merging can be incorporated into concolic testing of sequential applications. However, in order to support state merging for full-stack JavaScript web applications, it is necessary to define state merging for event-driven applications too. In sequential code, the state explosion problem is caused only by conditional branching nodes, which are created for branch conditions such as `if` statements. Testing of event-driven code, however, also introduces event branching nodes (cf. Section 4.4.1) that increase the number of program paths.

6.3.1 State Explosion in Event-driven Applications

Listing 6.4 presents a code snippet that registers three event handlers `b0`, `b1`, and `b2` with the `click` event. A partial symbolic execution tree for this snippet is depicted in Figure 6.8.

```
let b0Flag = false, b1Flag = false;
function b0() {
  b0Flag = true;
}
function b1() {
  b1Flag = true;
}
function b2() {
  if (b0Flag) {
    if (b1Flag) {
      foo();
    } else { ... }
   } else { ... }
   ...
 }
 document.getElementById("Button0").addEventListener("click", b0);
 document.getElementById("Button1").addEventListener("click", b1);
 document.getElementById("Button2").addEventListener("click", b2);
```

Listing 6.4: An event-driven code snippet of which the execution gives rise to a large number of program paths.

Event branching nodes of the form E_i appear in a path whenever the tester can decide to invoke a new event handler, the i-th handler in the event sequence, by triggering its corresponding event. Since event handlers in JavaScript are atomic, this decision mostly arises whenever the test executor has finished executing the previous handler. The very first event branching node is created when the test executor has finished executing all code outside of event handlers. In the case of Listing 6.4, no code is executed outside of any event handler apart from the code for registering these event handlers, so the root state of the tree is the event branching node E_0.

Every event branching node in this tree has four child states, as the tester can choose to execute any of the three registered event handlers, or it can decide to stop the test run by not executing any event handler. Even without accounting for conditional branching inside the event handlers themselves, the number of reachable program paths grows exponentially in function of the length of the event sequence. If state merging would be restricted to merging duplicate branching

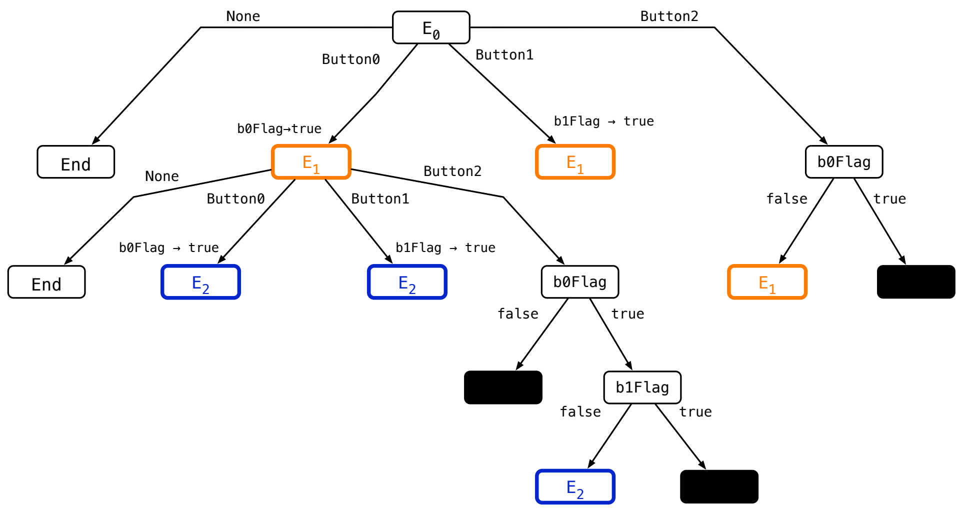


Figure 6.8: A partial symbolic execution tree for the code snippet in Listing 6.4. For brevity, the figure includes only one subtree for an E_1 event branching node, while the subtrees for all E_2 states are omitted entirely.

nodes together, state explosion *within* event handlers would be prevented, but the total number of states in the DAG would still remain exponential as the event branching nodes, along with their subtrees, would still be duplicated.

Consider for example line 14 in the snippet. For every invocation of event handler `b2`, the state corresponding to this line will be duplicated in the tree, as an instance of this state may be encountered after having executed the `else` branch of the `if` statement on line 9, or after having executed first the `then` branch on line 10 and then either the `then` branch or the `else` branch. The `mergeState` function of Algorithm 3 can be used to merge these three duplicate states together, but this would only reduce the number of states within one particular invocation of the event handler.

State merging should therefore not be limited to merging duplicate states *within* event handlers together. Many of these event branching nodes can be considered duplicates of each other: for every i, every event branching node of the form E_i is a duplicate of another. Duplicate occurrences of the event branching node E_1 are coloured orange in Figure 6.8, while duplicates of E_2 are coloured blue.

6.3.2 Considerations for State Merging in Event-driven Code

To apply state merging to event-driven code, we must solve the following problems:

- The program point must be redefined to include the event handler that was executed when the state was created.
- The event branching nodes encountered along the path towards a state must be translated into symbolic expressions, so that these expressions may appear in the predicate of ITE expressions. This enables constructing ITE expressions for variables whose value depends on the chosen event handler.

Redefining Program Points

Recall that the definition of a program point determines which states should be considered duplicates of each other, and which hence should be considered as candidate states for merging. Program points were defined in Section 6.2.1 as tuples of a code label, specifically the line number, and the function stack that was active when the state was observed.

There are two reasons for extending this definition to include event handlers. First, as a technical complication, because of the inclusion of the code label, the program point is tied to a particular location in the code. It is therefore not possible to express a program point for event branching nodes, as these states are created whenever the tester has finished executing an event handler and is preparing to invoke the next handler. Furthermore, because event handlers are atomic in JavaScript, the function stack for such a state would always be empty, as the JavaScript process cannot invoke a new event handler while another is still being executed. Since no code label can be included and the state's function stack would always be empty, we instead define a new type of program point `HandlerFinished(i)`, which we abbreviate to `HF(i)`. This state is used exclusively for event branching nodes, and expresses the point where the test executor has finished executing the i-th handler in the event sequence. A state of the form E_i receives the program point `HF(i - 1)`.

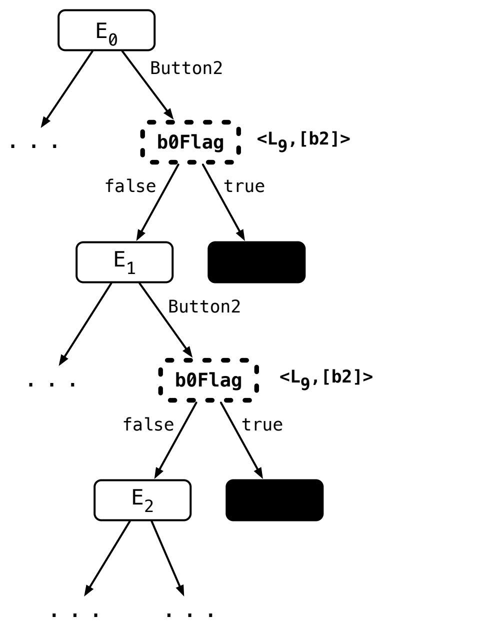
Second, it is necessary to distinguish between similar states of *different* invocations of the *same* event handler. Consider Figure 6.9a, which depicts a partial symbolic execution tree for Listing 6.4 without any merged states. The tree includes two instances of a `bOFlag` conditional branching node. One of these instances is created when handler `b2` is invoked a first time, the other is created when it is invoked the second time. The program point for both states is defined as $\langle L_9, [\texttt{b2}]\rangle$. As both states share the same program point, they could be merged together, which would result in the symbolic execution DAG in Figure 6.9b. However, doing so would introduce a loop in the symbolic execution DAG. DAGs are acyclic, so by introducing a loop, the resulting graph would no longer be a DAG. Not only does this complicate the traversal of the symbolic execution graph, but it also renders it possible for states to feature path constraints that have an infinite length.

At the same time, however, we do not want to make the program point overly restrictive. For example, including in the program point the entire stack of event handlers that have been called so far would make the program point far too specific, as it would become impossible to ever merge states from the subtrees of different handlers. These problems can be avoided by distinguishing between different invocations of the same handler, i.e., by including in the program point tuple the global *index in the event sequence* of the current event handler invocation. This index does not refer to the specific event handler being invoked, but rather describes its position in the event sequence. It hence does not keep track of a history of which event handlers were invoked previously, thereby avoiding the problem of the program point being too restrictive. The program point for the top `b0Flag` state of Figure 6.9a therefore becomes $\langle L_9, [\texttt{b2}], 1\rangle$, while that of the bottom one becomes $\langle L_9, [\texttt{b2}], 2\rangle$. Since the states now have different program points, the merging algorithm does not merge them together and the problem is avoided. If no event handler is currently being executed, we replace this index with the value `null`.

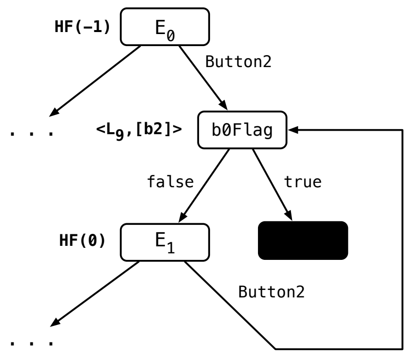
We therefore define a program point for event branching nodes as a `HF(i)` construct, and for conditional branching nodes as a triple consisting of the code label, the function stack, and either the index of the current event handler invocation or the value `null`.

Event Branching Nodes as Symbolic Expressions

As with sequential applications, the values of program variables may depend on the specific path that was followed. For example, the values of variables `b0Flag` and `b1Flag` depend on whether respectively the event handlers `b0` or `b1` have been executed at least once.



(a) A partial symbolic execution tree for Listing 6.4 with subtrees for handler `b0` expanded.



(b) Merging the duplicate states of the DAG together results in a loop.

Figure 6.9: An example of why it is necessary to include the current position in the event sequence as part of the program point. The program points are emphasised in bold.

Recall that when merging two separate states S_1 and S_2, the symbolic value for a program variable `x` in the merged state is replaced by an ITE expression of the form $ITE(PC_1, \sigma_1[\mathrm{x}], \sigma_2[\mathrm{x}])$, where PC_1 is the path constraint for S_1, and σ_1 and σ_2 are the symbolic stores for S_1 and S_2 respectively. The predicate values of such an ITE expression hence consist of the path constraint that lead to the merged states.

For sequential applications, a path constraint consists of the concatenation of the symbolic constraints corresponding to the conditional branching nodes encountered along that path. In the case of event-driven applications, however, the path is also determined by the event handlers that were executed. The path constraint must hence also encode the decision of which event handler was invoked when, which in turn means that the decision of which edge to take from an event branching node must be translated to a symbolic constraint.

This is possible by assigning a unique id to every event handler, named the *target id*, and to express event branching nodes of the form E_i as a constraint over a unique symbolic input variable e_i, named the *event input*. By following a branch from an event branching node, an event input is constrained to a specific target id. Event handlers `b0`, `b1`, and `b2` can be assigned respectively the target ids 0, 1, and 2, based on the order in which they were registered. Following the `Button1` edge for event branching node E_0, which corresponds to clicking on `Button1` and invoking event handler `b1` as the first event handler, is represented as the symbolic constraint $e_0 = 1$. As another example of this translation, the path constraint for the `b0Flag` state on the third row of Figure 6.8, which can be reached by first clicking `Button0` and then `Button2`, is $e_0 = 0 \wedge e_1 = 2$.

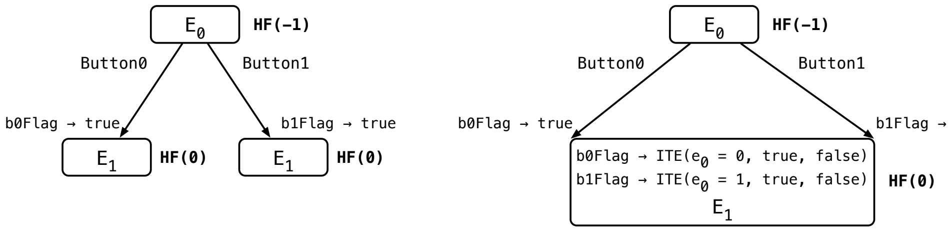


Figure 6.10: An example of how the program variables `b0Flag` and `b1Flag` are assigned different values along different paths in the unmerged symbolic execution tree (left) and the merged DAG (right). The program points are emphasised in bold.

Having this symbolic representation of decisions on event branching nodes makes it possible to construct ITE expressions for program variables whose value depends on the chosen event handler. Consider Figure 6.10, which depicts a small part of both the unmerged symbolic execution tree for Listing 6.4 and the merged DAG. In the merged DAG, the symbolic value for `b0Flag` can be expressed as $ITE(e_0 = 0, \texttt{true}, \texttt{false})$. If this symbolic value were to appear in another constraint, then the SMT solver can compute a mapping of event inputs to target ids to create an event sequence.

6.3.3 Result of State Merging in Event-driven Code

After taking these considerations into account, the symbolic execution DAG generated by a concolic tester that incorporates state merging for the event-driven code of Listing 6.4 is depicted in Figure 6.11.

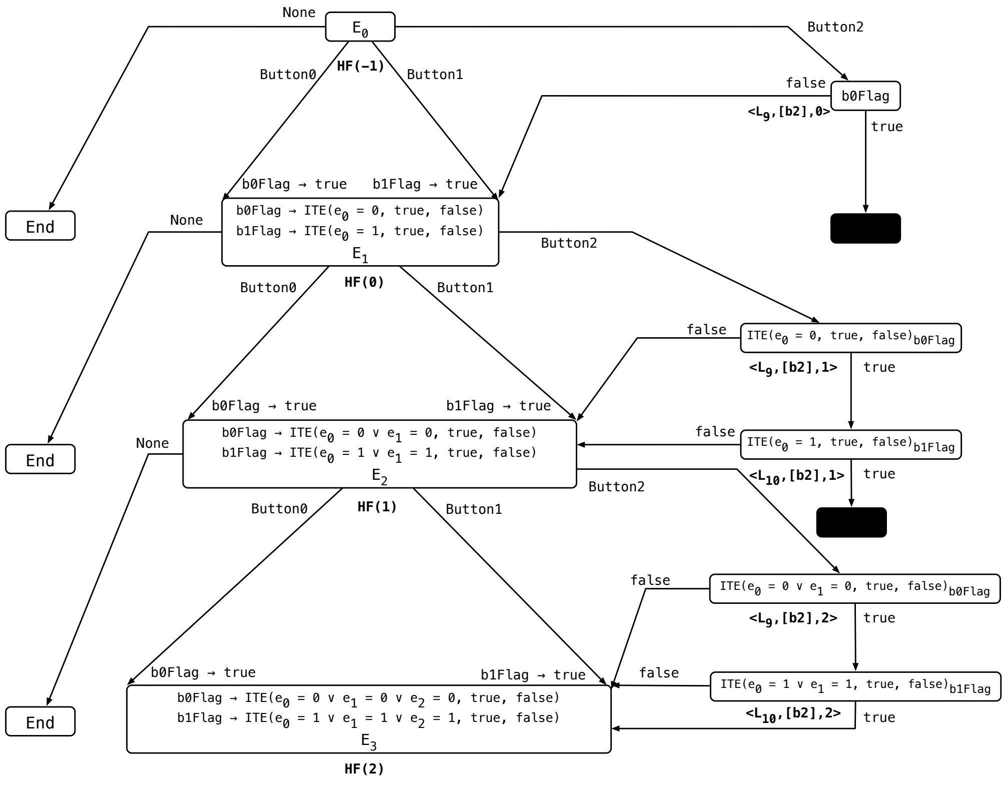

None
E0
HF(-1)
Button0
Button1
Button2
false
b0Flag
<L9,[b2],0>
true
End
None
b0Flag → true
b1Flag → true
b0Flag → ITE(e0 = 0, true, false)
b1Flag → ITE(e0 = 1, true, false)
E1
HF(0)
Button0
Button1
Button2
false
ITE(e0 = 0, true, false)b0Flag
<L9,[b2],1>
true
false
ITE(e0 = 1, true, false)b1Flag
<L10,[b2],1>
true
End
None
b0Flag → true
b1Flag → true
b0Flag → ITE(e0 = 0 ∨ e1 = 0, true, false)
b1Flag → ITE(e0 = 1 ∨ e1 = 1, true, false)
E2
HF(1)
Button0
Button1
Button2
false
ITE(e0 = 0 ∨ e1 = 0, true, false)b0Flag
<L9,[b2],2>
true
false
ITE(e0 = 1 ∨ e1 = 1, true, false)b1Flag
<L10,[b2],2>
true
End
b0Flag → true
b1Flag → true
b0Flag → ITE(e0 = 0 ∨ e1 = 0 ∨ e2 = 0, true, false)
b1Flag → ITE(e0 = 1 ∨ e1 = 1 ∨ e2 = 1, true, false)
E3
HF(2)

Figure 6.11: The symbolic execution DAG depicted in Figure 6.8 after applying state merging, with the states' program points emphasised in bold.

The DAG includes only one instance of an event branching node for every invocation of an event handler. Decisions over which event handler to invoke when, which affects the values of variables `b0Flag` and `b1Flag`, have been translated to symbolic constraints so that the symbolic value of these variables can be defined as ITE expressions.

Note that in the first two invocations of the event handler `b2`, the call to `foo` on line 11 is not reachable. In the first invocation of that event handler, the `then` branch of the conditional branching node `b0Flag` is unreachable, as this variable can only receive the value `true` if handler `b0` had been invoked first. In the second invocation, the `then` branch of the equivalent conditional branching node $\mathit{ITE}(e_0 = 0, \texttt{true}, \texttt{false})_{\texttt{b0Flag}}$ is reachable, but the `then` branch of $\mathit{ITE}(e_0 = 1, \texttt{true}, \texttt{false})_{\texttt{b1Flag}}$ is not reachable, since both ITE expressions cannot evaluate to `true` in the program execution.

The `true` branch of the `if` statement on line 10 at last becomes reachable in the third invocation of event handler `b2`, since a satisfiable mapping of event inputs to target ids (either $e_0 \rightarrow 0$ and $e_1 \rightarrow 1$ or vice versa) can be found so that both ITE expressions $\mathit{ITE}(e_0 = 0 \vee e_1 = 0, \texttt{true}, \texttt{false})$ and $\mathit{ITE}(e_0 = 1 \vee e_1 = 1, \texttt{true}, \texttt{false})$ can evaluate to `true`.

6.4 Implementation in StackFul

In this section, we describe how the state merging algorithm for event-driven programs presented in the previous sections has been implemented for StackFul. Although the implementation broadly follows the `mergeState` function defined in Algorithm 3, it does not keep a symbolic store for states in the symbolic execution DAG, nor does it explicitly model states as triples of the program point, path constraint, and symbolic store. StackFul instead lazily computes all required components of a state when necessary (i.e., when performing a new merge operation). We also describe some technical limitations in StackFul's approach to state merging.

6.4.1 Recomputing State Merging Tuples

The general outline for performing state merging (cf. Section 6.1) called for modelling each state of the symbolic execution tree as a tuple consisting of the program point, path constraint, and symbolic store. A merge of two states $\langle P, PC_1, \sigma_1\rangle \sim \langle P, PC_2, \sigma_2\rangle$ produces the merged state $\langle P, PC_1 \vee PC_2, \sigma_m\rangle$, with σ_m defined as $\sigma_m[v] = \mathit{ITE}(PC_1, \sigma_1[v], \sigma_2[v])$ for each variable v in σ_1. Recall that a suitable definition for program points ensures that states with identical program points also share the same lexical environment, so that their symbolic stores include the same set of variables. Although this definition of a merge operation allows for a conceptually simple approach to state merging, explicitly representing every state in the symbolic execution DAG as such a tuple is not practical because of the associated memory overhead.

However, it is not necessary to store each of these three elements for every state in the symbolic execution DAG. Of these three elements, StackFul only stores the program point for each state. Both the path constraint and the state's symbolic store are computed when necessary, as explained below. Additionally, the predicates of ITE expressions do not consist of the entire path constraint of a state.

Computing the Path Constraint

As conditional and event branching nodes can be translated into symbolic constraints, the path constraint of a state can be reconstructed by descending from the root to this state while concatenating the constraints of the states observed along this path. Recall that in the conceptual overview of state merging on state tuples (cf. Section 6.1.3), the path constraint of a merged state is the disjunction $PC_1 \vee PC_2$ of the path constraints of the two original states. These disjuncted path constraints also appear implicitly in the implementation because of the fact that in a merged symbolic execution DAG there may be several paths from the root to a state. The complete path constraint for a state can then be computed by taking the disjunction of all paths from the root to that state.

Creating ITE Expressions

Section 6.1.3 explained how ITE expressions are created when merging program variables with different values. If the variable `x` is assigned the values `1`, `2`, and `3` along the path constraints PC_1, PC_2, and PC_3 respectively, then its associated ITE expression might be $ITE(PC_1, 1, ITE(PC_2, 2, 3))$. In practice, however, the increased length and complexity of these three path constraints also increase the memory required to store each ITE expression, resulting in an unacceptable memory footprint for the tester. StackFul therefore does not include entire path constraints in the predicates of ITE expressions, but it differentiates between values of the ITE based on their common ancestor constraint.

Consider the code in Listing 6.5.

The complete path constraint leading to the value `1` being assigned to `x` on line 4 is $i_0 = 0 \wedge i_1 = 1$, and that for the value `2` on line 6 is $i_0 = 0 \wedge i_1 \neq 1$. The state merging approach described in Section 6.1.3 might represent the complete ITE expression for `x` as $ITE(i_0 = 0 \wedge i_1 = 1, 1, ITE(i_0 = 0 \wedge i_1 \neq 1, 2, ITE(i_0 \neq 0 \wedge i_2 = 2, 3, 4)))$. However, as the path constraints to the values `1` and `2` share the $i_0 = 0$ conjunct, we can distinguish between these based only on their common

```
1  let x;
2  if (randomInt() === 0) {
3    if (randomInt() === 1) {
4      x = 1;
5    } else {
6      x = 2;
7    }
8  } else {
9    if (randomInt() === 2) {
10      x = 3;
11    } else {
12      x = 4;
13    }
14 }
```

Listing 6.5: A code snippet that assigns four different values to `x`.

ancestor constraint $i_1 = 1$ (corresponding to the `if` statement at line 3). The ITE expression that distinguishes between `1` and `2` can then be expressed as $ITE(i_1 = 1, 1, 2)$.

We hence minimise the predicates of ITE expressions by articulating these only in terms of common ancestor constraints. The complete ITE expression for `x` is represented as $ITE(i_0 = 0, ITE(i_1 = 1, 1, 2), ITE(i_2 = 2, 3, 4))$.

Computing the Symbolic Store

Storing an explicit snapshot of the symbolic store for every state in the symbolic execution DAG results in a large memory overhead. Furthermore, the symbolic store for a state only becomes relevant when the state is involved in a merge operation. Rather than having the symbolic store for every state in the DAG remain in memory it is preferable to recompute the symbolic store for a state when necessary.

StackFul makes it possible to recompute a state's symbolic store by annotating the edges in the symbolic execution DAG with *store updates*. These annotations describe which store operations were observed by the test executor in the program between two constraints. The symbolic stores in StackFul that are recomputed in this manner are represented as a stack of *frames*, with each frame mapping a set of variables to their symbolic values. When execution enters a new scope, a new frame is pushed onto the stack. When execution leaves a scope, the topmost frame is popped.

StackFul employs the following types of store annotations:

Assignment updates take the form `VarAssign(varName, symExp)`. They specify that the program variable `varName` has been assigned the symbolic expression `symExp`.

Scope entry updates take the form `EnterScope(varNames)`, where `varNames` is a set of names of program variables. They model a push on the stack of frames and are generated when program execution enters a new scope in which the given identifiers enter in scope.

Scope exit updates take the form `ExitScope` and are generated when execution exits a scope.

An edge annotation consists of a, possibly empty, sequence of these updates. Figure 6.12 illustrates how the test executor annotates an edge in the symbolic execution tree with store updates when observing changes to the store.

When merging two states together, it is no longer necessary to construct a symbolic store for the merged state. Instead, the store annotations for the subtree below the merged state must be updated to account for this merge. The merge operation requires the symbolic values of assignment expressions in `VarAssign` annotations to be updated. This is illustrated in Figure 6.13, which depicts the symbolic execution tree of Figure 6.12 after merging the states $3 = 3$ and $4 = 3$ together. The `VarAssign` annotation in the `then` child of the merged state which before the merger mapped variable `y` to 3 now maps this variable to $ITE(i_0 = 1, 3, 4)$.

6.4.2 Technical Limitations of the Prototype Implementation

Having presented an overview of how state merging is implemented in StackFul, we now list two technical limitations that limit the scope of state merging, namely the merging of heap locations and merging that takes place inside a loop. Apart from specifying the technical challenge raised by these issues, we also describe potential solutions to these problems.

```
let x = randomInt(), y = 0, z = 0;
if (x === 1) {
  let x = 2;
  y = 3;
} else {
  y = 4;
}
if (y === 3) {
  z = x + y;
}
```

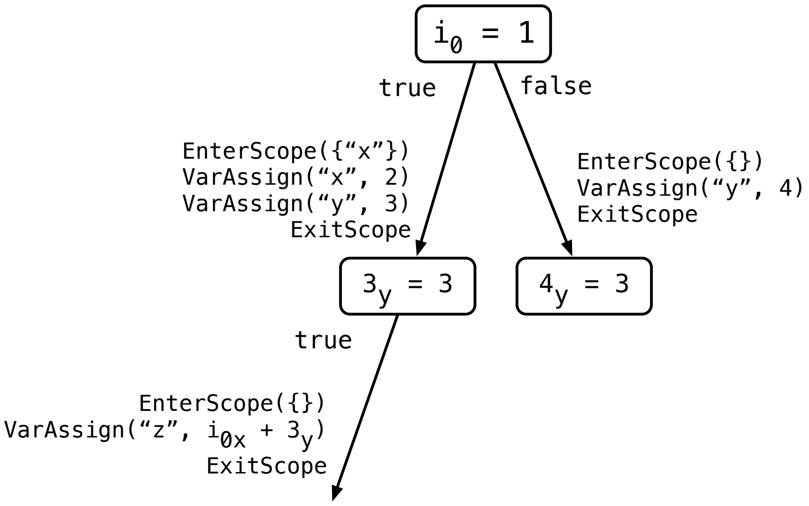


Figure 6.12: An example of how updates to the symbolic store can be expressed via edge annotations.

Heap Locations

One main limitation to the current implementation of state merging in StackFul lies in how state merging handles the merging of *heap locations*. Heap locations are used in symbolic execution when e.g., creating an object, or reading from or writing to a field of an object [5, 39]. In symbolic execution, objects are stored in a symbolic heap $\mathcal{H}$, where they can be accessed via a heap location (i.e., an address). The symbolic heap is defined as a mapping of locations to objects, where an object is defined as a mapping of fields to values. When a new object is created, a new, unique heap location is created and the symbolic heap is extended with an entry that maps the location to the object.

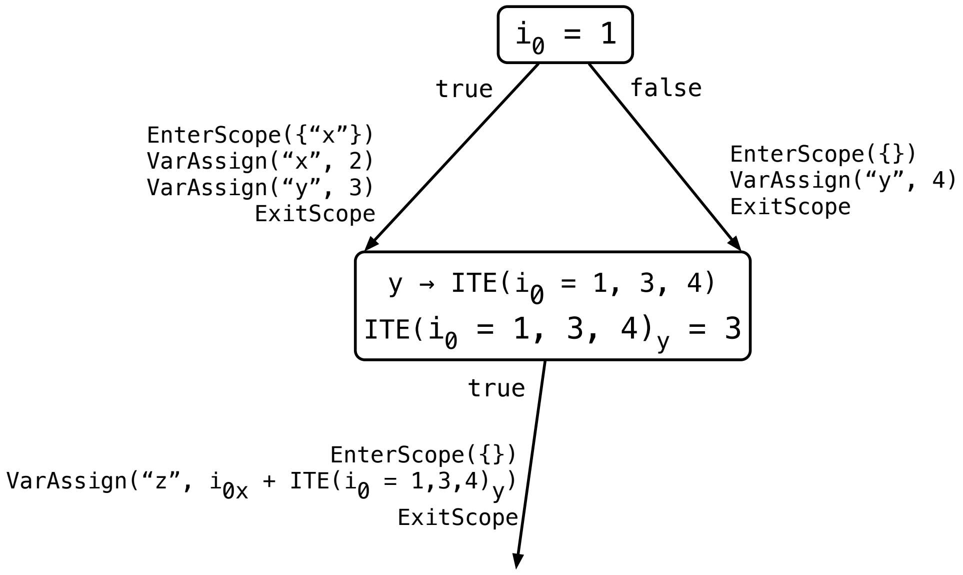


Figure 6.13: The `VarAssign` annotation along the `then` edge below the merged state has been updated to reflect the changed value for `y`.

For example, after executing the definition statement `let o = {x:1, y:2}`, a new symbolic heap location ℓ is created, the symbolic heap is extended with the entry $\ell \rightarrow \{\mathtt{x} \rightarrow 1, \mathtt{y} \rightarrow 2\}$, and the program variable `o` receives the symbolic heap location ℓ. Performing a read operation on an object's field, e.g., `o.x`, can then be performed symbolically by looking up the heap location to which `o` is bound in the symbolic heap, and then reading the field `x` from the resulting object. A write operation, e.g., `o.x = 42`, can be performed by first applying a similar lookup procedure and then overwriting the value for the field `x` of the object.

Consider now the code snippet in Listing 6.6. After merging both program paths after the first `if` statement, the program variable `o3` is conditionally aliased with either the heap location of `o1` or that of `o2`, and should hence be represented symbolically as an ITE expression over these two locations. However, when reading the field `x` on line 7, the aforementioned lookup procedure can no longer be applied, as `o3` does not point to a unique location.

StackFul currently does not offer a solution to this problem. Applying state merging as described in Section 6.2 on two states where a program variable is

```
let o1 = {x: 1}, o2 = {x: 2}, o3;
if (randomInt() === 1) {
  o3 = o1;
} else {
  o3 = o2;
}
if (o3.x === 1) { ... } else { ... }
```

Listing 6.6: A code snippet which results in merging of heap pointers.

conditionally bound with separate objects is hence unsound. However, we sketch a, currently unimplemented, solution in which we define a read operation from or a write operation to an object field of an ITE expression.

We represent a read operation on a field `f` via a symbolic expression X as $\mathcal{H}[X][\mathtt{f}]$, which is defined as follows:

$$\mathcal{H}[X][\mathtt{f}] = \begin{cases} ITE(c, \mathcal{H}[\ell_t][\mathtt{f}], \mathcal{H}[\ell_e][\mathtt{f}]) & \text{if } X \text{ evaluates to } ITE(c, \ell_t, \ell_e) \\ o[\mathtt{f}] & \text{if } X \text{ evaluates to a heap location } \ell \text{ and} \\ & \mathcal{H} \text{ contains the mapping } \ell \rightarrow o \end{cases}$$

For simplicity, we do not model the case where $\mathcal{H}$ does not contain an entry for ℓ. Note, however, that such a scenario would result in an exception being thrown at run time.

A write operation of the value v on a field `f` via a symbolic expression X is represented as $\mathcal{H}[X][\mathtt{f}]$ `:=` v, and updates the heap in the following way when executed:

$$\begin{cases} \mathcal{H}[\ell_t][\mathtt{f}] \;:=\; ITE(c, v, v_{t\_old}) \text{ and} & \text{if } X \text{ evaluates to } ITE(c, \ell_t, \ell_e)\text{, with } v_{t\_old} \\ \mathcal{H}[\ell_e][\mathtt{f}] \;:=\; ITE(c, v_{e\_old}, v) & \text{and } v_{e\_old} \text{ defined as the previous value} \\ & \text{for respectively } \mathcal{H}[\ell_t][\mathtt{f}] \text{ and } \mathcal{H}[\ell_e][\mathtt{f}] \\ \mathcal{H}[\ell][\mathtt{f}] \;:=\; v & \text{if } X \text{ evaluates to } \ell \end{cases}$$

As before, we do not model the case where $\mathcal{H}$ does not contain an entry for ℓ.

Apart from redefining the read operation and write operation, incorporating this solution into StackFul would also require the edges of the symbolic execution DAG to be annotated with *heap updates*. These updates are analogous to the store updates described in Section 6.4.1. Furthermore, it will be necessary to support object constraints (cf. Section 4.2.3) to reason over conditional pointer aliasing.

Merging inside Loops

Another limitation of the state merging algorithm described in this chapter relates to the unrolling of loops. Recall that in the context of event-driven applications, a program point is defined as a triple of the code label currently being executed, i.e., the line number, the function stack, and the index in the event sequence of the current event handler invocation. However, this does not account for code that is executed inside a program loop, where the same code label is observed multiple times using the same function stack and event handler invocation. Note that this is not a problem for recursive functions as, without tail recursion optimisation, recursive calls to a function are saved in the function stack of the program point. Different invocations of a recursive function hence correspond to different stacks.

Consider the code in Listing 6.7, where a loop with a fixed number of iterations is executed as part of the event handler for a button click. Without state merging, this loop gives rise to the creation of 1024 new paths in every invocation of the event handler. State merging is therefore necessary to prevent the number of states from exploding.

However, as the program point for the `if` statement on line 4 is identical across the ten loop iterations, our state merging algorithm would, in each loop iteration, attempt to merge the conditional branching node that was created in the current iteration with that of the previous iteration. This would introduce a cycle into the symbolic execution DAG, as the state now features a back-edge to itself.

```
1 document.getElementById("button")
2   .addEventListener("click", function () {
3   for (let i = 0; i < 10; i++) {
4     if (randomInt() === 0) { ... } else { ... }
5   }
6 });
```

Listing 6.7: A program loop inside an event handler.

Although currently not implemented, the problem can be avoided entirely by including the concrete loop iteration in the program point. Merging may then still take place across the same iteration, i.e., the `then` branch and `else` branch of the conditional branching node point to the same state, thereby avoiding the state explosion problem. In effect, this corresponds to unrolling the loop before state merging, similar to existing approaches [12, 66].

6.5 Evaluation

In this section, we evaluate the state merging approach on the set of eight input programs previously presented in Section 5.5. The incorporation of state merging into StackFul$_{\text{INTER}}$ stands orthogonal to the two-phase approach for distinguishing between high priority and low priority server-side errors described in Chapter 5. This evaluation therefore focuses on measuring the effect of state merging on inter-process testing of full-stack JavaScript web applications. That is, we run two versions of StackFul$_{\text{INTER}}$, a baseline version that does not incorporate state merging and another that does which we name StackFul, on the eight input programs. Rather than first running an intra-process phase followed by an inter-process phase, both testers are configured to test the entire application in only a single inter-process phase. When testing the applications, both testers instantiate a single client process and a single server process. In all applications, the client process serves as the entry point into the application for the tester.

6.5.1 Overview of the Evaluation

We evaluate StackFul$_{\text{INTER}}$ and StackFul on the eight input programs previously presented in Section 5.5. We compare StackFul$_{\text{INTER}}$ with StackFul, which is identical to StackFul$_{\text{INTER}}$ except for the fact that it incorporates the state merging algorithm as described in Section 6.4.

We evaluate the effectiveness of state merging through the following research questions:

RQ1 Does state merging improve the code coverage of the tester per test run?

RQ2 Does state merging increase the computational overhead per test run?

RQ3 Does state merging offer an increase in coverage over the same unit of execution time? That is, does state merging make it possible to cover a larger part of the application in less time?

State merging promises to reduce the number of unexplored program states in an application's symbolic execution tree, thereby increasing code coverage more quickly as there are fewer states to explore that correspond with the same lexical position in the code. However, state merging also greatly increases the burden on the SMT solver, thereby leading to longer solve times compared to the baseline approach. The aim of research question 1 is hence to verify whether state merging achieves a higher code coverage per test run, when leaving aside the increased execution times per test run. Research question 2 investigates to what extent the increased burden on the SMT solver increases execution time per test run. As a hypothetical example, even a tenfold increase in solving time may constitute only a fraction of the total execution time per test run. Research question 3 combines both investigations by directly asking the question of whether state merging achieves a higher code coverage over the same span of execution time.

We answer these three research questions through the following metrics:

Line coverage We measure the number of lines of code that were executed at least once in any test run, expressed as a fraction of the total lines of code in the application.

Branch coverage We measure the total number of branch conditions of which both the `then` branch and the `else` branch were explored at least once [90].

Event handler coverage As a last coverage metric, we also measure how many registered event handlers were invoked at least once.

Execution time We measure the overhead induced by state merging when testing an application by observing the execution time required by StackFul and StackFul$_{\text{INTER}}$ to complete the same number of test runs.

Specifications

The test executor of both testers, implemented in TypeScript, uses Node.js v18.7.0, while the test selector, implemented in Scala, uses Java SE Runtime Environment build 19.0.1+10-21, configured to run with a maximum heap size of 4GB. Both testers were executed on a 2.8 GHz Quad-Core Intel Core i7 CPU, with 16GB of 2133 MHz LPDDR3 RAM, running macOS 13.2. As an SMT solver, both testers use Z3, version 4.8.5 - 64bit.

6.5.2 RQ1: Code Coverage per Test Run

The first research question asks whether, and to what extent, state merging improves the code coverage achieved by the tester per test run. To answer this question, StackFul and StackFul$_{\text{INTER}}$ are allocated the same *test run budget*, and we measure the coverage they achieve within this budget. The number of test runs to be completed is not fixed across all applications because the duration of a single test run heavily depends on the application itself. Using the same, fixed number of test runs would therefore result in testing of some applications finishing quickly, whereas other applications would not be finished in any reasonable timespan. We therefore find a test run per application by having StackFul test an application for 3600 seconds and observing both the coverage level it obtains and the number of test runs it completes. StackFul$_{\text{INTER}}$ is then allocated the same test run budget. StackFul and StackFul$_{\text{INTER}}$ test an application in a deterministic manner. The code coverage that they achieve over a series of test runs is therefore also deterministic. Hence, it suffices to have both testers test each application only once.

To inspect the code coverage obtained by both testers, we employ the line coverage and branch coverage metrics. Because of the event-driven nature of the full-stack JavaScript web applications, we also use an event handler coverage which measures the fraction of registered event handlers which have been invoked at least once.

Line Coverage

The results of the line coverage metric are shown in Figure 6.14. The X axis specifies how many test runs have been completed, and the Y axis specifies the fraction (presented as a percentage value) of the total lines of code in the application that have been executed at least once. StackFul$_{\text{INTER}}$ is represented by the orange line with triangles, and StackFul is represented by the blue line with circles.

The results indicate that the line coverage achieved by StackFul is generally higher than or equal to that of StackFul$_{\text{INTER}}$: it achieves a higher coverage for four out of eight applications and an ultimately equal coverage for the other four. We observe that for some applications, StackFul slightly lags behind StackFul$_{\text{INTER}}$, although, for the eight applications that we evaluated, StackFul eventually catches up with StackFul$_{\text{INTER}}$. Across all applications, StackFul shows an increase in line coverage of -0.8% to 8.2% over StackFul$_{\text{INTER}}$ at the end of the testing session.

Line coverage is generally high, especially considering some parts of the server are unreachable from that client of the application. After manual inspection of the covered lines, we determined that the remaining uncovered lines for Simple Chat were unreachable from the client. TOHacks also includes a fairly large number of unreachable lines, with many overlapping branch conditions.

In the case of the Chat application, the uncovered lines relies on an operation that is unsupported by the symbolic executor. The concolic tester may therefore not be able to steer execution towards both branches of the branch condition that features the unsupported operation. State merging also does not achieve an improvement in line coverage for Game of Life. This application features several nested loops. As mentioned in Section 6.4.2, StackFul currently does not allow for merging inside program loops.

The highest increase in line coverage is attained for the Calculator application (cf. Section 2.1.2), which notably registers a large number of event handlers. In order to reach the server's `compute` message handler (cf. Section 2.1.2), a relatively complex sequence of events must be triggered on the client. State merging not only reduces the number of test runs required by StackFul to explore the individual event handlers, it also allows for finding the appropriate sequence of events to access the server's message handler more quickly. The Slack Mockup similarly requires triggering a somewhat complex sequence of client-side events in order to reach certain parts of the server.

The Totems and Whiteboard applications both feature relatively intricate constraints on user input, where malformed user inputs lead to early termination of the program. State merging manages to explore both branches of the user input validation more quickly than the baseline version. However, because the branches that reject user input feature relatively few lines of code, this results in only a modest improvement in line coverage.

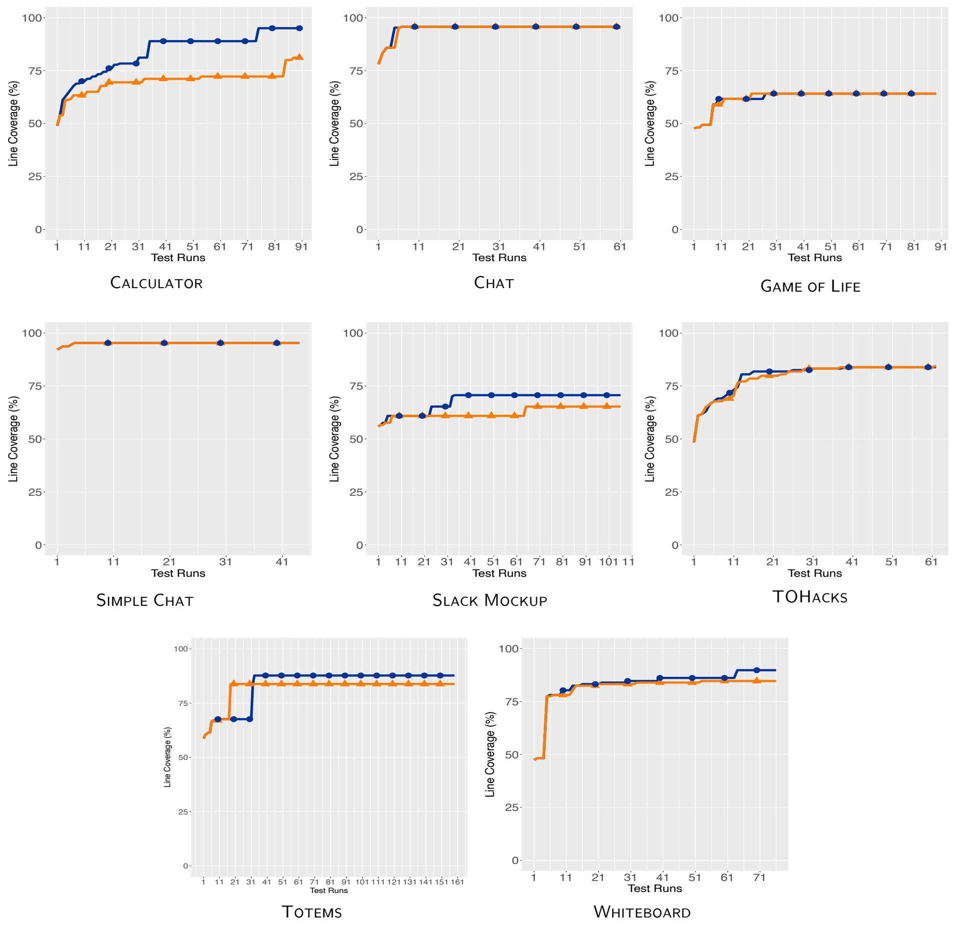


Figure 6.14: Line coverage achieved by the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

Branch Coverage

As an alternative measure of coverage for sequential applications, we also observe branch coverage obtained by both testers on the eight applications. The results are depicted in Figure 6.15. As expected, these results mostly align with the line coverage results. StackFul achieves a higher or equal branch coverage, though it again sometimes slightly lags behind StackFul$_{\text{INTER}}$ for a few intermediate test runs. Across all applications, StackFul shows an increase in branch coverage of -6.7% to 28.6% over StackFul$_{\text{INTER}}$ at the end of the testing session.

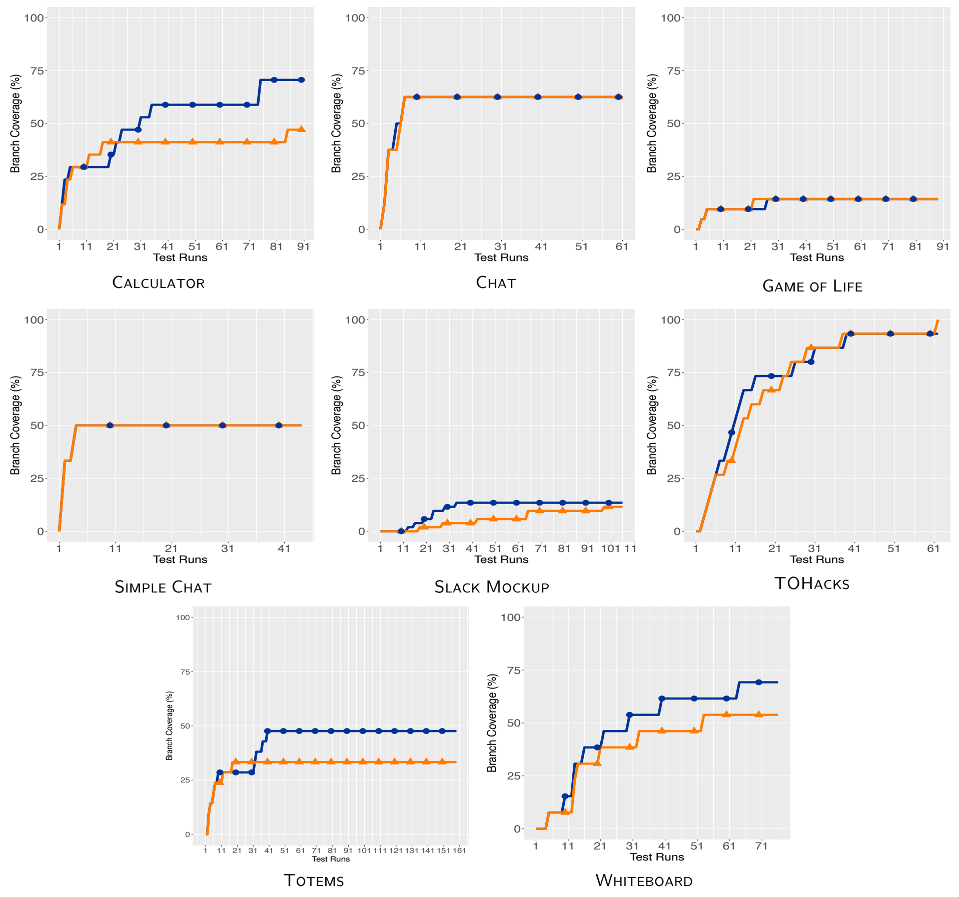


Figure 6.15: Branch coverage achieved by the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

Event Handler Coverage

As full-stack JavaScript web applications are event-driven, we include an additional coverage metric which measures the fraction of event handlers that were invoked at least once. The results are depicted in Figure 6.16.

Event handler coverage is generally very high: both testers eventually achieve 100% for most applications. StackFul$_{\text{INTER}}$ does not achieve full event handler coverage in Calculator, however, whereas StackFul achieves 100% coverage in relatively few test runs. By default, StackFul and StackFul$_{\text{INTER}}$ use a breadth-first search strategy for exploring event handlers, which tends to favour event handlers that are registered early in the application over handlers that are registered later. This proves detrimental to event handler coverage when testing applications that feature many event handlers, such as CALCULATOR. STACKFUL, however, overcomes this obstacle via the use of state merging.

Across all applications, STACKFUL shows an increase in event handler coverage of 0.0% to 66.7% over STACKFUL$_{\text{INTER}}$ at the end of the testing session.

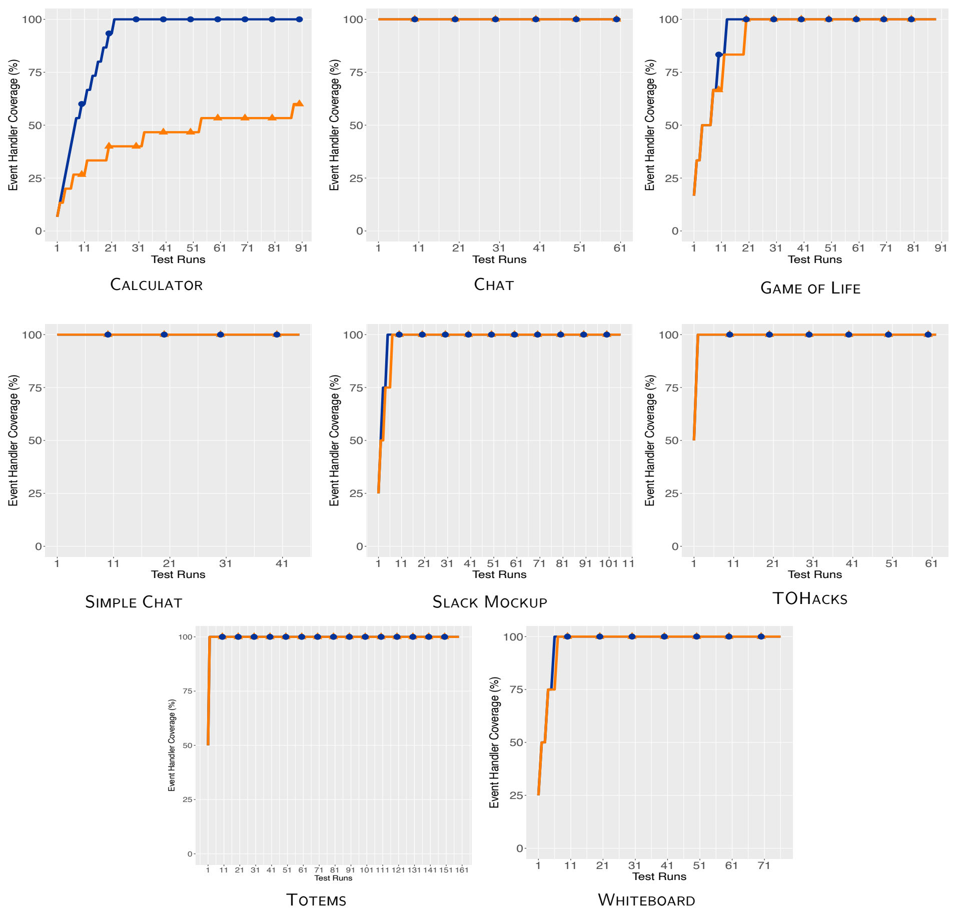


Figure 6.16: Portion of event handlers explored at least once by the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

Conclusion

The three coverage metrics indicate that StackFul attains a higher or equal level of code coverage in fewer test runs. We conclude that state merging improves the code coverage achieved by the tester per test run, and we hence answer the first research question in the affirmative.

6.5.3 RQ2: Computational Overhead per Test Run

The second research question concerns the computational overhead of employing state merging. It asks whether incorporating state merging increases the duration required to complete a test run. We answer this question by observing the execution time required to complete a number of test runs. As before, due to the large variations in execution time across all applications, we do not employ a fixed number of test runs. We run StackFul for 5400 seconds (1.5 hours) and record the number of test runs it has completed. Afterwards, we allocate the same test run budget to StackFul$_{\text{INTER}}$.

Since we answer this research question by investigating the execution time of both testers, which is inherently non-deterministic, we let both StackFul and StackFul$_{\text{INTER}}$ test each application five times, and we take the median of their execution times. In practice, the differences in execution times per iteration are very small, relative to the total time budget of 5400 seconds.

Execution Time

Figure 6.17 depicts the execution times required by both testers to complete their test run budget. We observe for all applications that state merging increases the execution time required to complete the same number of test runs. We also observe that the gap in execution time between both variants increases over time. We attribute this to the SMT solver requiring progressively more time to solve path constraints that grow increasingly complex. Furthermore, as more merge operations have taken place, the number of ITE expressions in these path constraints also grows. Solving these ITE expressions is known to be particularly computationally intensive for SMT solvers [66].

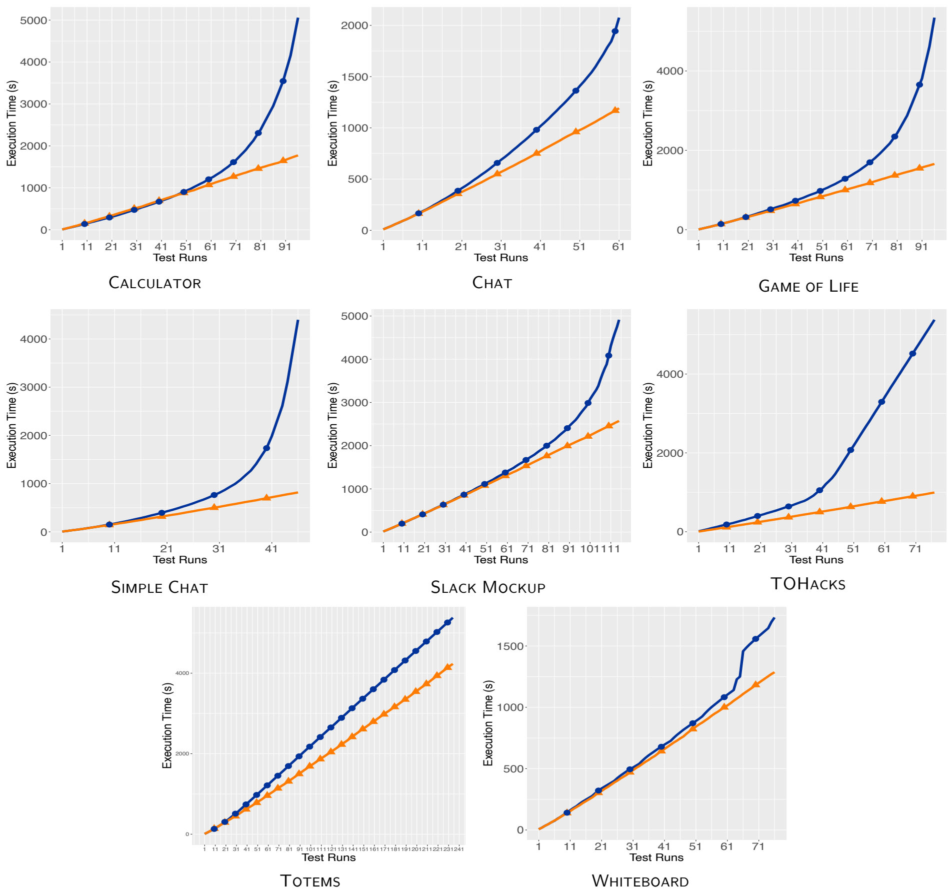


Figure 6.17: Execution time (in seconds) required by the tester for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

Across all applications, StackFul shows an increase in execution time of 27.1% to 440.5% over StackFul$_{\text{INTER}}$ at the end of the testing session. We conclude that state merging has a detrimental effect on the execution time per test run because it increases the computational overhead of the tester per test run. We hence answer the second research question in the affirmative.

6.5.4 RQ3: Code Coverage per Unit of Execution Time

We have concluded that state merging generally increases the code coverage attained by StackFul (RQ1), but also increases the execution time of each test run (RQ2). Our third research question combines both elements by asking directly whether state merging increases the code coverage attained by the tester over a unit of execution time, or whether a baseline tester achieves a higher coverage by virtue of the fact that it completes more test runs over the same time interval.

We answer this research question by measuring code coverage in function of the execution time. Rather than allocating a budget of test runs to both testers, we run both for 5400 seconds (1.5 hours) while recording the code coverage that they attain. As the most general of the three coverage types previously employed, we measure only line coverage. As with the second research question, we let StackFul$_{\text{INTER}}$ and StackFul test each application five times and then take the median of the execution times.

Line Coverage over Execution Time

Figure 6.18 depicts the line coverage attained by both testers in function of their execution times. Note that execution times are expressed on a log scale, as line coverage generally increases more quickly towards the beginning of a test execution than towards the end. As before, we find that the line coverage attained by StackFul is generally higher than or equal to that of StackFul$_{\text{INTER}}$. Despite the higher number of test runs completed by StackFul$_{\text{INTER}}$ (cf. RQ2), StackFul is still generally superior with regards to code coverage. Across all applications, StackFul shows an increase in line coverage over execution time of -0.8% to 4.7% over StackFul$_{\text{INTER}}$ at the end of the testing session.

We conclude that state merging generally offers an increased code coverage compared to StackFul$_{\text{INTER}}$ over the same unit of execution time, and hence answer the third research question in the affirmative as well.

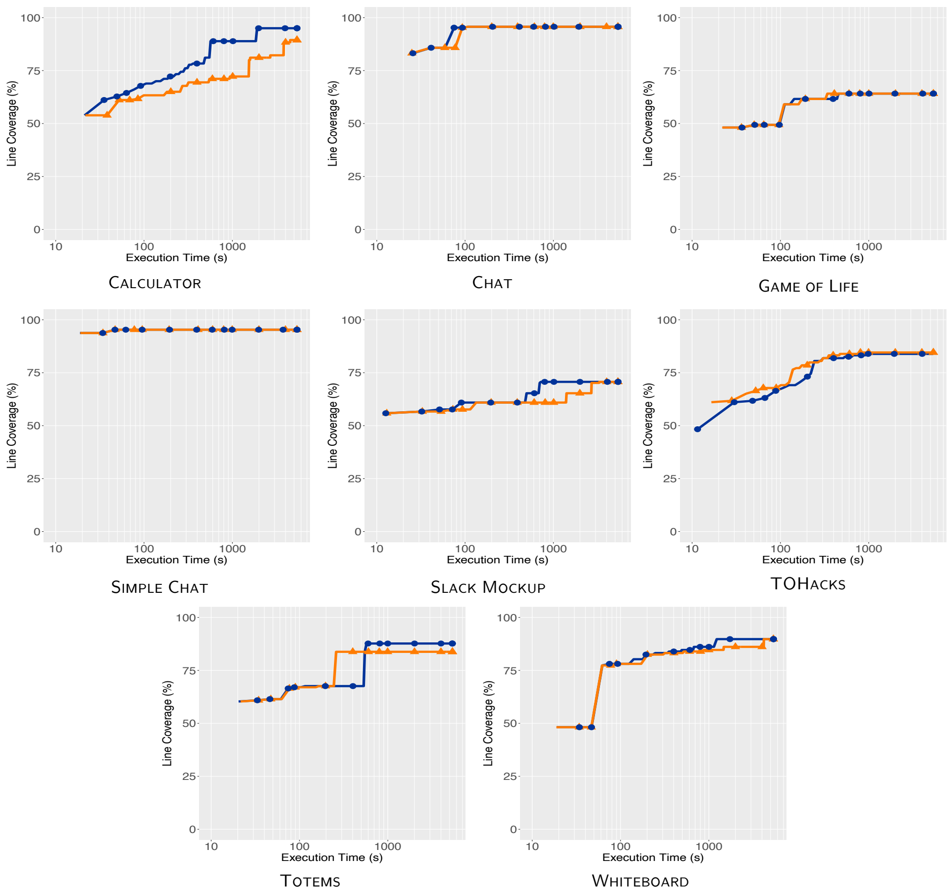


Figure 6.18: Line coverage attained by the tester in function of the execution time (in seconds) for eight input programs, with (blue, with circles) and without (orange, with triangles) state merging applied.

6.6 Conclusion

In this chapter, we described an approach to incorporating *state merging* in concolic testing of event-driven code. This algorithm allows for merging similar states together, hence curbing the state explosion problem that arises during naive testing. This is the first approach that enables state merging to be performed in the setting of concolic testing of event-driven applications, which in turn enables state merging for concolic testing of full-stack JavaScript web applications. We summarise the chapter and conclude by comparing StackFul against the criteria outlined in Section 2.4 for testing full-stack JavaScript web applications.

6.6.1 Summary

Naive concolic testing gives rise to the state explosion problem, where the number of states to be covered by the tester grows exponentially in function of the number of branch conditions that are executed. One approach for mitigating this problem is to merge similar states together, thereby reducing the total number of states in the symbolic execution DAG. When merging states, shared variables that have been assigned different values in each state can be represented as a single variable with an ITE expression as value.

We distinguished between state merging performed by *online* symbolic executors, and *offline* executors, such as concolic testers. Although state merging for concolic testers is more cumbersome than for online testers because of the path determinacy problem, we described an approach where offline state merging mimics the state merging performed by online testers. We also specified how state merging can be incorporated into concolic testing of event-driven applications, by extending the definition of a *program point* to incorporate event sequences and by translating event branching nodes into symbolic expressions that can be incorporated into ITE constructs. We described a prototype implementation for this approach to state merging. We also discussed the prototype's current limitations and their potential solutions.

We evaluated StackFul, which incorporates state merging, and StackFul$_{\text{INTER}}$, which does not, on eight full-stack JavaScript web applications. We compared both testers with respect to the code coverage that they achieve per test run, the computational overhead of state merging per test run, and the code coverage per unit of execution time. Our evaluation finds that StackFul achieves a higher code coverage for the same number of test runs, as well as for the same length of execution time on about half of the applications, while it attains an equal coverage on the other applications.

Table 6.1: StackFul evaluated on the criteria identified in Section 2.4.

Criterion		Description	Satisfied?
①	A	Capable of testing sequential JavaScript code	✓
	B	Models inputs from the DOM	✓
②	A	Allows for dynamic (de)registration of event handlers	✓
	B	Explores event space	✓
③		Capable of finding composition-specific faults	✓
④	A	Whole-program monitoring	✓
	B	Observes communication between distributed processes	✓

6.6.2 Concluding Remarks

We conclude the chapter by evaluating StackFul on the criteria for concolic testing of full-stack JavaScript web applications identified in Section 2.4. Recall that StackFul$_{\text{INTER}}$ satisfied six of the seven criteria (cf. Section 5.6): all except for the efficient exploration of the application's event space (criterion 2.B).

Table 6.1 evaluates StackFul on these criteria. State merging does not extend or affect the capabilities already achieved by StackFul$_{\text{INTER}}$ but rather stands orthogonal to the inter-process procedure employed by this tester. It therefore does not interfere with the criteria already achieved by StackFul$_{\text{INTER}}$. However, by alleviating the state explosion problem, we enable a more efficient exploration of the application's search space, thereby resulting in the compliance of criterion 2.B. By satisfying this last criterion, we hence conclude that StackFul meets all criteria for concolic testing of full-stack JavaScript web applications.

7 Conclusion

This dissertation presented *StackFul*, a concolic tester for full-stack JavaScript web applications. StackFul uses inter-process testing to improve precision and prevent false-positive errors from being reported. StackFul also incorporates state merging to alleviate the state explosion problem. The source code of StackFul is publicly available at `https://github.com/softwarelanguageslab/StackFul` In this final chapter, we summarise our work, revisit our contributions, and discuss avenues for future research.

7.1 Summary

Chapter 2 defined full-stack JavaScript web applications as web applications of which both the client and the server processes are implemented in JavaScript. Their unique set of characteristics gives rise to four challenges: the dynamic nature of JavaScript, the necessity to test event-driven code, the need for handling different multiplicities, and the interplay between the application's constituent processes. These challenges were distilled into seven concrete criteria for having automated testers test these applications.

Chapter 3 presented the state of the art in automated testing of distributed systems, event-driven applications, JavaScript programs, and web servers. We evaluated the most relevant concolic testers for web clients and servers on the seven criteria that were identified in the previous chapter.

Chapter 4 presented *StackFul$_{INTRA}$*, an intra-process concolic tester for JavaScript. StackFul$_{\text{INTRA}}$ instruments the JavaScript code under test so it can perform shadow execution. This enables the tester to execute the code while simultaneously collecting symbolic constraints over the program inputs. StackFul$_{\text{INTRA}}$ is also capable of testing event-driven programs, by observing which event handlers are registered and deregistered and then automatically triggering the corresponding user event or system event. As an intra-process tester, StackFul$_{\text{INTRA}}$ only tests a single process and relies on mocking of messages to cover the corresponding message handler. Nevertheless, we evaluated the tester on the seven criteria and found that it satisfies three of them.

Chapter 5 introduced the concept of inter-process testing and implemented this technique in *StackFul$_{\text{INTER}}$*. We described how StackFul$_{\text{INTER}}$ simultaneously tests multiple processes of the full-stack JavaScript web application, how it intercepted and observed the communication between these processes, and how it managed a global, cross-process path constraint. As a motivation for inter-process testing, StackFul$_{\text{INTER}}$ was used to distinguish between high-priority and low-priority server errors, based on whether a server error is reachable from a client in the application. We evaluated StackFul$_{\text{INTER}}$ on three research questions related to this use case. We then evaluated StackFul$_{\text{INTER}}$ on the seven criteria and found that it satisfies six of them.

Chapter 6 discussed how the state explosion problem — which arises when testing even sequential, single-process code but is made worse in the context of event-driven, multi-process applications — can be alleviated by merging similar states together. We showed how state merging can be applied by online symbolic executors and concolic testers on sequential code, and then lifted the technique to concolic testing of event-driven code. We implemented this form of state merging in *StackFul*, and evaluated the effectiveness of StackFul in covering full-stack JavaScript web applications. Finally, we evaluated StackFul on the seven criteria and found that it satisfies all of them.

7.2 Revisiting the Contributions

This dissertation presented the following four contributions:

1. **Identifying Challenges for Automated Testing of Full-stack JavaScript Web Applications:** We identified the following challenges for automated testing of full-stack JavaScript web applications:

    - As all constituent processes of the full-stack JavaScript web application are implemented in JavaScript, the tester must be capable of handling the **dynamic nature of JavaScript**.

   - Since both client and server processes are **event-driven**, each registering event handlers and message handlers, the tester must be capable of generating and testing sequences of events and messages.

   - As the number of instances of clients and servers is not fixed, and because certain bugs may only manifest themselves when a specific number of processes have connected, the tester must be capable of **handling different multiplicities**.

   - The tester must consider the **interplay between processes**, as well as how execution of one process affects that of others.

2. **Inter-process Concolic Testing:** We defined inter-process testing as *testing the composition of all processes in a full-stack JavaScript web application as a whole, while observing their communication, and having their execution paths cross process boundaries.* Inter-process testing was implemented in StackFul$_{\text{INTER}}$, which was built on top of StackFul$_{\text{INTRA}}$.

   Chapter 4 discussed how StackFul$_{\text{INTRA}}$ performs concolic testing of event-driven JavaScript applications, and hence how both StackFul$_{\text{INTRA}}$ and StackFul$_{\text{INTER}}$ tackle the first and second challenge. Chapter 5 described how StackFul$_{\text{INTER}}$ is capable of launching and testing several processes simultaneously, thus addressing the third challenge. Chapter 5 also stipulated how StackFul$_{\text{INTER}}$ is capable of observing message sends and message receives, and how this enables constructing precise global path constraints that cross process boundaries. This in turn helps StackFul$_{\text{INTER}}$ solve the fourth challenge.

3. **Distinguishing between High-Priority and Low-Priority Server Errors:** The increased precision offered by inter-process testing enabled StackFul$_{\text{INTER}}$ to distinguish between high-priority and low-priority server errors, based on whether the server error was reachable from a client process in the application. StackFul$_{\text{INTER}}$ employed a two-phase testing approach to perform this classification. In the first phase, StackFul$_{\text{INTER}}$ performs traditional, intra-process testing of the server to construct path constraints from the entry points of the server, e.g., message handlers, to these errors. In the second phase, StackFul$_{\text{INTER}}$ performs inter-process testing, and starts testing the application from its clients. StackFul$_{\text{INTER}}$ then searches for paths that cause a client to send a message to the

server that results in the previously detected server error to be triggered again, hence marking it as a high-priority error. Errors that could not be reproduced in this manner were labelled as low-priority errors.

4. **State Merging for Concolic Testing of Event-driven Applications:** When testing an application, the number of distinct execution paths, or states, that are available in the application generally grows exponentially in function of the number of branch conditions encountered while testing the application. This problem is exacerbated for i) event-driven programs, as event handlers introduce new branching with conditions, and ii) inter-process testing as more branching points may be encountered on longer, cross-process execution paths

   This problem can be alleviated by applying state merging, in which similar states are merged together to reduce the number of states. We discussed how state merging can be performed for concolic testers like StackFul$_{\text{INTER}}$, and further lifted this technique to the domain of concolic testing of event-driven application. We then incorporated this technique into our tester, naming it StackFul.

7.3 Limitations and Future Work

We envision several avenues for future research.

7.3.1 State Merging for Two-phase Inter-process Testing

Recall that the two-phase approach to inter-process testing consists of an intra-process phase which tests just the server, followed by an inter-process testing phase where the entire application is tested via its clients. In the first phase, the server's message handlers are used as entry points into the execution of the server. For every server error detected by the tester, the tester stores the path constraints from the message handler to the error. In the second phase, the tester aims to reach the previously detected server errors from the clients of the application. To this end, when testing the clients in the second phase, the tester looks for message sends where the client sends a message that is received by a message handler which was previously reported as a possible entry point for a server error. The tester then attempts to match the current path constraint, ranging from the client's starting point to the message send operation, with the previously reported path constraint from the message handler to the server error. If a match is satisfiable, the server error can be reproduced and the error is marked as a high-priority error. Any server errors remaining at the end of the inter-process testing phase are deemed unreachable and labelled as low-priority errors.

Although StackFul incorporates state merging into inter-process testing, it has not yet been integrated with this two-phase approach to inter-process testing. Applying state merging to the first phase would allow for increased code coverage, and hence would increase the number of server errors that can be found within the same test budget. Likewise, incorporating state merging into the second phase would allow the tester to more quickly find points in the execution where the client communicates with the server, and hence where a server error may be reached. However, applying state merging would also complicate both types of path constraints. It may then not be feasible for the SMT solver to determine whether both path constraints can be matched.

7.3.2 Heuristical Search for Inter-process Testing

Heuristical search strategies (cf. Section 3.5.1) may prioritise exploration of program paths that reach previously uncovered parts of the code [20], or program paths that maximise a specific metric, such as the number of read-write conflicts between event handlers [72]. In the context of inter-process testing, one could employ such a heuristic for directing testing towards message sends in the code where the application's constituent processes communicate with each other. This would allow the tester to more quickly find points in the program where the execution of one process may affect that of another. However, heuristical search strategies may also interfere with the application of state merging, since state merging prefers for testing to traverse the application's control flow in a topological order (i.e., exhaustively exploring a node in a control-flow graph before moving on to the exploration of its descendants), to maximise the opportunities for merging states [66].

Nevertheless, developing a heuristic which allows the tester to more quickly find points of contact between the application's processes while respecting the preferred order in which state merging drives execution may increase the code coverage attained by the tester.

7.3.3 Automatic Exploration of Multiplicities

Although StackFul is capable of testing various multiplicities of an application, the desired multiplicity must be configured by StackFul's user. That is, the user must specify the number of processes to be started, their type, and the order in which they should be launched. Once StackFul has been launched, this configuration remains fixed throughout all test runs.

A more flexible approach would be to have StackFul investigate automatically which multiplicities should be explored, i.e., which multiplicities give rise to different, potentially buggy, behaviour of the application. We can leverage research undertaken in the context of testing actor programs or distributed systems, where automated testers must also consider the number and type of processes to be spawned. For example, COMPI [73] is a concolic tester for message-passing applications that renders the number of processes symbolic. This is sufficient for scenarios where a process directly queries the number of processes so that this number is included directly in the program's path constraint. TAP [75] investigated a more general approach, where the tool performs backwards symbolic execution on actor programs. This allows TAP to reason over the number of actors that must be spawned and how their message schedules must be constructed in order to reach a particular targets statement in the code.

7.3.4 Scaling StackFul to Test Larger Applications

Although StackFul has been evaluated on real-world full-stack JavaScript web applications, the scale of the investigated applications was rather limited. There remain several practical obstacles that prevent the tester from scaling to larger applications:

1. Even when merging similar program states, the state explosion problem remains an important challenge. Although StackFul is capable of removing almost all duplicates of a state, state merging introduces such a large overhead on the SMT solver that the timed gained by preventing duplicate states from being explored is largely offset by the increased execution time per test run. It may be advantageous to only apply state merging conditionally, when the impact of merging on the SMT solver can be estimated to be sufficiently low [66].

2. StackFul requires more sophisticated heuristics for exploring applications that are highly event-driven: i.e., applications that require a large number of events to be triggered before reaching a specific part of the application. For example, StackFul finds it difficult to test parts of a website that are only made available when a user has logged in to the website, as these sites typically require many different navigation events before allowing the user access. To solve this problem, we might employ *path prefixes* [14]: programmer-defined sequences of events that are guaranteed to lead the tester to exercise a desired part of the application. A more general solution would consist of employing new search heuristics to test event-driven applications, potentially taking into account the synergy between these heuristics and state merging (cf. Section 7.3.2).

3. StackFul does not support web workers, which are often employed on modern web applications to improve the performance on multicore machines. Web workers interleave non-deterministically and may lead to race conditions that cannot be reproduced by the tester. The problem of testing multithreaded applications has been well studied in existing work [115, 53, 67, 127]. There is hence a wide body of known techniques for testing concurrent systems that can be incorporated into StackFul.

4. The execution overhead imposed by the instrumented code is fairly high, so that StackFul experiences a noticeable slowdown when testing even sequential JavaScript applications. It may prove beneficial to configure the instrumentation library to remove redundant calls to the advice (cf. Section 4.2.1).

5. A memory model should be added to StackFul to allow for reasoning over array operations and object constraints without having to resort to concretisation.

7.4 Closing Remarks

Concolic testing of full-stack JavaScript web applications gives rise to several challenges. First, the event-driven nature of these applications forces concolic testing to generate sophisticated sequences of user events and system events to completely cover the application. Second, these applications are implemented in JavaScript, which is notoriously difficult to test. Third, these applications do not consist of a fixed number of instances of client and server processes. Fourth, because of the interconnected nature of these processes, the execution of one process may affect that of another in unpredicable ways.

In light of these challenges, we demonstrated that *intra-process testing*, where a tester only tests a single process in isolation from others, is no longer sufficient. We introduced *inter-process testing*, which we defined as *testing the composition of all processes in a full-stack JavaScript web application as a whole, while observing their communication, and having their execution paths cross process boundaries.* We demonstrated how to perform inter-process testing on these applications, and how this leads to an increase in the precision of the tester. This increased precision allowed us to distinguish between high-priority and low-priority server errors, based on whether the server error was reachable from a client in the application.

However, inter-process testing also exacerbates the state explosion problem, where the number of states that an application may assume grows exponentially in function of the branching points encountered while testing the application. To alleviate the state explosion problem, we detailed how state merging can be integrated into concolic testing of event-driven applications. We implemented this form of state merging into a concolic tester for inter-process testing, and named the resulting tester *StackFul*.

We believe that these contributions are the first steps towards the wider application of inter-process concolic testing.

Bibliography

[1] Abdullah, H. M. and Zeki, A. M. (2014). Frontend and Backend Web Technologies in Social Networking Sites: Facebook as an Example. In *3rd International Conference on Advanced Computer Science Applications and Technologies*, pages 85–89. IEEE.

[2] Afzal, W., Torkar, R., and Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, **51**(6), 957–976.

[3] Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K. (2010). A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering*, **36**(6), 742–762.

[4] Alshahwan, N., Gao, X., Harman, M., Jia, Y., Mao, K., Mols, A., Tei, T., and Zorin, I. (2018). Deploying Search Based Software Engineering with Sapienz at Facebook. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, pages 3–45.

[5] Anand, S. and Harrold, M. J. (2011). Heap Cloning: Enabling Dynamic Symbolic Execution of Java Programs. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 33–42.

[6] Anand, S., Godefroid, P., and Tillmann, N. (2008). Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 367–381.

[7] Anand, S., Naik, M., Harrold, M. J., and Yang, H. (2012). Automated Concolic Testing of Smartphone Apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 59.

[8] Anand, S., Burke, E. K., Chen, T. Y., Clark, J. A., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and McMinn, P. (2013). An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal of Systems and Software*, **86**(8), 1978–2001.

[9] Andreasen, E., Gong, L., Møller, A., Pradel, M., Selakovic, M., Sen, K., and Staicu, C. (2017). A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, **50**(5), 66:1–66:36.

[10] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A. M., and Ernst, M. D. (2010). Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*, **36**(4), 474–494.

[11] Artzi, S., Dolby, J., Jensen, S. H., Møller, A., and Tip, F. (2011). A Framework for Automated Testing of JavaScript Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 571–580.

[12] Avgerinos, T., Rebert, A., Cha, S. K., and Brumley, D. (2016). Enhancing Symbolic Execution with Veritesting. *Communications of the ACM*, **59**(6), 93–100.

[13] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. (2018). A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, **51**(3).

[14] Bardin, S., Kosmatov, N., Marre, B., Mentré, D., and Williams, N. (2018). Test Case Generation with PathCrawler/LTest: How to Automate an Industrial Testing Process. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, pages 104–120.

[15] Barrett, C., Deters, M., De Moura, L., Oliveras, A., and Stump, A. (2013). 6 Years of SMT-COMP. *Journal of Automated Reasoning*, **50**(3), 243–277.

[16] Böhme, M., Pham, V., Nguyen, M., and Roychoudhury, A. (2017). Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344.

[17] Böhme, M., Pham, V., and Roychoudhury, A. (2019). Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, **45**(5), 489–506.

[18] Bounimova, E., Godefroid, P., and Molnar, D. A. (2013). Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 122–131.

[19] Buchanan, M. (2015). Physics in Finance: Trading at the Speed of Light. *Nature*, **518**(7538), 161–163.

[20] Burnim, J. and Sen, K. (2008). Heuristics for Scalable Dynamic Test Generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 443–446.

[21] Cadar, C. and Sen, K. (2013). Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, **56**(2), 82–90.

[22] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. (2006). EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 322–335.

[23] Cadar, C., Dunbar, D., and Engler, D. R. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224.

[24] Camps, U. S. B. (2020). What Is a Full Stack Developer & What Do They Do? `https://bootcamp.learn.utoronto.ca/blog/what-is-a-full-stack-developer`, Last accessed on 2022-05-12.

[25] Cha, S. and Oh, H. (2019). Concolic Testing with Adaptively Changing Search Heuristics. In *Proceedings of the ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pages 235–245.

[26] Cha, S., Hong, S., Lee, J., and Oh, H. (2018). Automatically Generating Search Heuristics for Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1244–1254.

[27] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394.

[28] Champion, M., Byrne, S., Nicol, G., and Wood, L. (1998). Document Object Model (Core) Level 1.

[29] Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., and Liu, W. (2018). A Systematic Review of Fuzzing Techniques. *Computers & Security*, **75**, 118–137.

[30] Chen, T., Zhang, X.-s., Guo, S.-z., Li, H.-y., and Wu, Y. (2013). State of the Art: Dynamic Symbolic Execution for Automated Test Generation. *Future Generation Computer Systems*, **29**(7), 1758–1773.

[31] Chipounov, V., Kuznetsov, V., and Candea, G. (2012). The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, **30**(1), 2:1–2:49.

[32] Christakis, M., Müller, P., and Wüstholz, V. (2016). Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 144–155, New York, NY, USA. ACM.

[33] Christophe, L. (2020). Aran. `https://github.com/lachrist/aran`, Last accessed on 2022-06-23.

[34] Christophe, L., Gonzalez Boix, E., De Meuter, W., and De Roover, C. (2016). Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pages 260–270. IEEE.

[35] Christophe, L., De Roover, C., Gonzalez Boix, E., and De Meuter, W. (2018). 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*.

[36] Clarke, L., Glendinning, I., and Hempel, R. (1994). The MPI Message Passing Interface Standard. In *Programming Environments for Massively Parallel Distributed Systems*, pages 213–218. Springer.

[37] Cousot, P. (1996). Abstract Interpretation. *ACM Computing Surveys*, **28**(2), 324–328.

[38] de Moura, L. M. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340.

[39] Deng, X., Lee, J., and Robby (2012). Efficient and Formal Generalized Symbolic Execution. *Automated Software Engineering*, **19**(3), 233–301.

[40] Derakhshanfar, P., Devroey, X., Perrouin, G., Zaidman, A., and van Deursen, A. (2020). Search-Based Crash Reproduction Using Behavioural Model Seeding. *Software Testing, Verification and Reliability*, **30**(3).

[41] Dowson, M. (1997). The Ariane 5 Software Failure. *ACM SIGSOFT Software Engingeering Notes*, **22**(2), 84.

[42] Earle, C. B. (2000). Symbolic Program Execution Using the Erlang Verification Tool. In *9th International Workshop on Functional and Logic Programming, WFLP'2000, Benicassim, Spain, September 28-30, 2000*, pages 42–55.

[43] ECMA (2021). ECMA-262, 12th Edition, June 2021 ECMAScript® 2021 Language Specification.

[44] Elfriede Dustin, Jeff Rashka, J. P. (1999). *Automated Software Testing: Introduction, Management, and Performance.* Addison-Wesley Professional.

[45] Express (2017). Using Middleware. `https://expressjs.com/en/guide/using-middleware.html`, Last accessed on 2022-05-27.

[46] Felleisen, M. and Friedman, D. P. (1987). A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 314, New York, NY, USA. Association for Computing Machinery.

[47] Ferles, K., Wüstholz, V., Christakis, M., and Dillig, I. (2017). Failure-Directed Program Trimming. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 174–185.

[48] Gay, G. (2017). The Fitness Function for the Job: Search-Based Generation of Test Suites That Detect Real Faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 345–355.

[49] Godefroid, P. (2007). Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 47–54, New York, NY, USA. Association for Computing Machinery.

[50] Godefroid, P. and Luchaup, D. (2011). Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 23–33.

[51] Godefroid, P., Levin, M. Y., and Molnar, D. A. (2008). Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*.

[52] Godefroid, P., Levin, M. Y., and Molnar, D. A. (2012). SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM*, **55**(3), 40–44.

[53] Guo, S., Kusano, M., Wang, C., Yang, Z., and Gupta, A. (2015). Assertion Guided Symbolic Execution of Multithreaded Programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 854–865.

[54] Hamill, M. and Goseva-Popstojanova, K. (2017). Analyzing and Predicting Effort Associated with Finding and Fixing Software Faults. *Information and Software Technology,*, **87**, 1–18.

[55] Hansen, T., Schachte, P., and Søndergaard, H. (2009). State Joining and Splitting for the Symbolic Execution of Binaries. In *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, pages 76–92.

[56] Harman, M. and McMinn, P. (2007). A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 73–83, New York, NY, USA. Association for Computing Machinery.

[57] Harman, M., Jia, Y., and Zhang, Y. (2015). Achievements, Open Problems and Challenges for Search Based Software Testing. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–12.

[58] Ivancic, F. (2020). SunDew: Systematic Automated Security Testing. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, page 3.

[59] Jaffar, J., Murali, V., and Navas, J. A. (2013). Boosting Concolic Testing via Interpolation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 48–58.

[60] Jensen, C. S., Prasad, M. R., and Møller, A. (2013). Automated Testing with Targeted Event Sequence Generation. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 67–77.

[61] Jensen, S. H., Jonsson, P. A., and Møller, A. (2012). Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 34–44.

[62] Kähkönen, K., Kindermann, R., Heljanko, K., and Niemelä, I. (2010). Experimental Comparison of Concolic and Random Testing for Java Card Applets. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*, pages 22–39.

[63] Khari, M. and Kumar, P. (2019). An Extensive Evaluation of Search-Based Software Testing: a Review. *Soft Computing*, **23**(6), 1933–1946.

[64] Kim, M., Kim, Y., and Jang, Y. (2012). Industrial Application of Concolic Testing on Embedded Software: Case Studies. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 390–399.

[65] Kuchta, T., Palikareva, H., and Cadar, C. (2018). Shadow symbolic execution for testing software patches. *ACM Trans. Softw. Eng. Methodol.*, **27**(3), 10:1–10:32.

[66] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. (2012). Efficient State Merging in Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 193–204.

[67] Kähkönen, K., Saarikivi, O., and Heljanko, K. (2013). LCT: A Parallel Distributed Testing Tool for Multithreaded Java Programs. *Electronic Notes in Theoretical Computer Science*, **296**, 253–259. Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC).

[68] Leveson, N. G. (2017). The Therac-25: 30 Years Later. *Computer*, **50**(11), 8–11.

[69] Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., and Jr., E. J. W. (2013). Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 372–381.

[70] Lewis, S. (2019). Definition: Full-stack Developer. `https://www.theserverside.com/definition/full-stack-developer`, Last accessed on 2022-05-12.

[71] Li, G. and Ghosh, I. (2013). PASS: String Solving with Parameterized Array and Interval Automaton. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pages 15–31.

[72] Li, G., Andreasen, E., and Ghosh, I. (2014). 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.

[73] Li, H., Li, S., Benavides, Z., Chen, Z., and Gupta, R. (2018a). COMPI: Concolic Testing for MPI Applications. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 865–874.

[74] Li, J., Zhao, B., and Zhang, C. (2018b). Fuzzing: a Survey. *Cybersecurity*, **1**(1), 6.

[75] Li, S., Hariri, F., and Agha, G. (2018c). Targeted Test Generation for Actor Systems. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 8:1–8:31.

[76] Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. (2018). Fuzzing: State of the Art. *IEEE Transactions on Reliability*, **67**(3), 1199–1218.

[77] Lin, Y., Miller, T., and Søndergaard, H. (2015). Compositional Symbolic Execution Using Fine-Grained Summaries. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, pages 213–222.

[78] Liu, Y., Zhou, X., and Gong, W.-W. (2017). A Survey of Search Strategies in the Dynamic Symbolic Execution. In *ITM Web of Conferences*, volume 12, page 03025. EDP Sciences.

[79] Liu, Z. and Gupta, B. (2019). Study of Secured Full-stack Web Development. In *Proceedings of 34th International Conference on Computers and Their Applications, CATA 2019, Honolulu, Hawaii, USA, March 18-20, 2019*, pages 317–324.

[80] Loring, B., Mitchell, D., and Kinder, J. (2017). ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 196–199.

[81] Loring, B., Mitchell, D., and Kinder, J. (2019). Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, PLDI 2019, page 425–438, New York, NY, USA. Association for Computing Machinery.

[82] Ma, K.-K., Phang, K. Y., Foster, J. S., and Hicks, M. (2011). Directed Symbolic Execution. In *International Static Analysis Symposium*, pages 95–111. Springer.

[83] Majumdar, R. and Sen, K. (2007). Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 416–426.

[84] Mamun, A. M. A. (2022). Full Stack. `https://www.webopedia.com/definitions/full-stack/`, Last accessed on 2022-05-12.

[85] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2019). The Art, Science, and Engineering of Fuzzing: a Survey. *IEEE Transactions on Software Engineering*, **47**(11), 2312–2331.

[86] Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., Mols, A., and Scott, A. (2019). SapFix: Automated End-To-End Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–278.

[87] McMinn, P. (2011). Search-Based Software Testing: Past, Present and Future. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 153–163.

[88] Miller, C., Peterson, Z. N., *et al.* (2007). Analysis of Mutation and Generation-Based Fuzzing. *Independent Security Evaluators*, **4**.

[89] MozillaSecurity (2016). GitHub - MozillaSecurity/funfuzz: A Collection of Fuzzers in a Harness for Testing the SpiderMonkey JavaScript Engine.

[90] Myers, G. J. (2011). *The Art of Software Testing (3. Edition)*. Wiley.

[91] Nethercote, N. and Seward, J. (2007). Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100.

[92] Noller, Y., Nguyen, H. L., Tang, M., Kehrer, T., and Grunske, L. (2019). Complete Shadow Symbolic Execution with Java PathFinder. *ACM SIGSOFT Softw. Eng. Notes*, **44**(4), 15–16.

[93] Nuseibeh, B. (1997). Ariane 5: Who Dunnit? *IEEE Software*, **14**(3), 15–16.

[94] Oberg, J. (1999). Why the Mars Probe Went Off Course [Accident Investigation]. *IEEE Spectrum*, **36**(12), 34–39.

[95] Ognawala, S., Ochoa, M., Pretschner, A., and Limmer, T. (2016). MACKE: Compositional Analysis of Low-Level Vulnerabilities With Symbolic Execution. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 780–785.

[96] Ognawala, S., Kilger, F., and Pretschner, A. (2019). Compositional Fuzzing Aided by Targeted Symbolic Execution. *CoRR*, **abs/1903.02981**.

[97] Park, J., Lim, I., and Ryu, S. (2016). Battles With False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 61–70.

[98] Park, S., Hossain, B. M. M., Hussain, I., Csallner, C., Grechanik, M., Taneja, K., Fu, C., and Xie, Q. (2012). CarFast: Achieving Higher Statement Coverage Faster. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 35.

[99] Pasareanu, C. S. and Visser, W. (2009). A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal on Software Tools for Technology Transfer*, **11**(4), 339–353.

[100] Patton, R. (2005). *Software Testing*. Sams Publishing, second edition.

[101] Prather, R. E. and Jr., J. P. M. (1987). The Path Prefix Software Testing Strategy. *IEEE Transactions on Software Engineering*, **13**(7), 761–766.

[102] Pustogarov, I., Ristenpart, T., and Shmatikov, V. (2017). Using Program Analysis to Synthesize Sensor Spoofing Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 757–770.

[103] Qi, D., Roychoudhury, A., and Liang, Z. (2010). Test Generation to Expose Changes in Evolving Programs. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 397–406.

[104] Rollbar (2021). The State of Software Code Report. `https://content.rollbar.com/hubfs/State-of-Software-Code-Report.pdf`, Last accessed on 2022-06-19.

[105] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., and Winter, C. (2015). Tricorder: Building a Program Analysis Ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608.

[106] Sampaio, G., Santos, J. F., Maksimovic, P., and Gardner, P. (2020). A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, pages 28:1–28:29.

[107] Santos, J. F., Maksimovic, P., Naudziuniene, D., Wood, T., and Gardner, P. (2018a). JaVerT: JavaScript Verification Toolchain. *Proceedings of the ACM on Programming Languages*, **2**(POPL), 50:1–50:33.

[108] Santos, J. F., Maksimovic, P., Grohens, T., Dolby, J., and Gardner, P. (2018b). Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 11:1–11:14.

[109] Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., and Wehrle, K. (2010). KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks, IPSN 2010, April 12-16, 2010, Stockholm, Sweden*, pages 186–196.

[110] Sasnauskas, R., Dustmann, O. S., Kaminski, B. L., Wehrle, K., Weise, C., and Kowalewski, S. (2011). Scalable Symbolic Execution of Distributed Systems. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 333–342.

[111] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. (2010a). A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528.

[112] Saxena, P., Hanna, S., Poosankam, P., and Song, D. (2010b). FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*.

[113] Scheurer, D., Hähnle, R., and Bubel, R. (2016). A General Lattice Model for Merging Symbolic Execution Branches. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings*, pages 57–73.

[114] Sen, K. and Agha, G. (2006a). 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.

[115] Sen, K. and Agha, G. A. (2006b). Concolic Testing of Multithreaded Programs and its Application to Testing Security Protocols. Technical report, UIUC.

[116] Sen, K., Kalasapur, S., Brutch, T. G., and Gibbs, S. (2013). Jalangi: a Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498.

[117] Sen, K., Necula, G. C., Gong, L., and Choi, W. (2015). MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 842–853.

[118] Seo, H. and Kim, S. (2014). How We Get There: A Context-Guided Search Strategy in Concolic Testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 413–424, New York, NY, USA. Association for Computing Machinery.

[119] Sharma, A., Patani, R., and Aggarwal, A. (2016). Software Testing Using Genetic Algorithms. *International Journal of Computer Science & Engineering Survey*, **7**(2), 21–33.

[120] Shih-Kun, H., Han-Lin, L., Wai-Meng, L., and Huan, L. (2013). CRAX-web: Automatic Web Application Testing and Attack Generation. In *IEEE 7th International Conference on Software Security and Reliability, SERE 2013, Gaithersburg, MD, USA, June 18-20, 2013*, pages 208–217.

[121] Shropshire, J., Landry, J. P., and Presley, S. S. (2018). Towards a Consensus Definition of Full-stack Development. In *Proceedings of the Southern Association for Information Systems Conference, St. Augustine, IL, USA*, pages 1–6.

[122] Song, J., Cadar, C., and Pietzuch, P. R. (2014). SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications. *IEEE Trans. Software Eng.*, **40**(7), 695–709.

[123] Spencer, B., Benedikt, M., Møller, A., and van Breugel, F. (2017). ArtForm: a Tool for Exploring the Codebase of Form-Based Websites. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 380–383.

[124] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

[125] Strejcek, J. and Trtík, M. (2012). Abstracting Path Conditions. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 155–165.

[126] Sun, K. and Ryu, S. (2017). Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Computing Surveys*, **50**(4), 59:1–59:34.

[127] Sun, X. (2023). *Concolic Testing of Programs with Concurrent Dynamic Data Structures*. University of California, Riverside.

[128] Tillmann, N. and de Halleux, J. (2008). Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153.

[129] Vandercammen, M., Christophe, L., Di Nucci, D., De Meuter, W., and De Roover, C. (2020). Prioritising Server Bugs via Inter-process Concolic Testing. *The Art, Science, and Engineering of Programming*, **5**(2), 5.

[130] Vidal, G. (2014). Towards Symbolic Execution in Erlang. In *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, pages 351–360.

[131] W3C (2023). DOM - Living Standard. `https://dom.spec.whatwg.org/#events`, Last accessed on 2023-06-20.

[132] W3Techs (2022). Usage Statistics of JavaScript as Client-side Programming Language on Websites. `https://w3techs.com/technologies/details/cp-javascript`, Last accessed on 2022-05-30.

[133] Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., and Wei, J. (2017). A Comprehensive Study on Real World Concurrency Bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 520–531.

[134] Wang, K., Wang, Y., and Zhang, L. (2014). Software Testing Method Based on Improved Simulated Annealing Algorithm. In *10th International Conference on Reliability, Maintainability and Safety (ICRMS)*, pages 418–421. IEEE.

[135] Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., and Lin, Y. (2018). Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 291–302.

[136] Wüstholz, V. and Christakis, M. (2020). Targeted Greybox Fuzzing With Static Lookahead Analysis. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 789–800.

[137] Xie, T., Tillmann, N., de Halleux, J., and Schulte, W. (2009). Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*, pages 359–368.

[138] Xie, X., Liu, Y., Le, W., Li, X., and Chen, H. (2015). S-looper: Automatic Summarization for Multipath String Loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 188–198.

[139] Xie, X., Chen, B., Liu, Y., Le, W., and Li, X. (2016). Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 61–72.

[140] Yang, G., Filieri, A., Borges, M., Clun, D., and Wen, J. (2019). Chapter Five - Advances in Symbolic Execution. *Advances in Computers*, **113**, 225–287.

[141] You, S., Findler, R. B., and Dimoulas, C. (2021). Sound and Complete Concolic Testing for Higher-order Functions. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, pages 635–663.

[142] Zheng, Y., Zhang, X., and Ganesh, V. (2013). Z3-str: a Z3-Based String Solver for Web Application Analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 114–124.