Imperial College London
Department of Computing

A Trusted Infrastructure for Symbolic Analysis of Event-based Web APIs

Gabriela Cunha Sampaio

March 2022

Supervised by Prof. Philippa Gardner and Prof. José Fragoso Santos

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Gabriela Cunha Sampaio

Copyright Declaration

Abstract

JavaScript has been widely adopted for the development of Web applications, being used for both client and server-side code. Client-side JavaScript programs commonly interact with Web APIs, for instance, to capture the user interaction with the Web page via events. The use of such APIs increases the complexity of JavaScript programs. In fact, most errors in these programs are caused by the misuse of Web APIs. There are several approaches for detecting errors in client-side JavaScript programs, but they either assume the use of a single API or do not model APIs faithfully, giving rise to inconsistent behaviour and lack of trust.

We address the problem by developing a trustworthy infrastructure for the static analysis of Web APIs. We focus on two aspects of JavaScript programs: *event-driven* and *message-passing* programming, as these paradigms are common sources of confusion among developers. We choose to target the DOM event model and the JavaScript Promises and JavaScript async/await, which facilitate event-driven programming. Additionally, we target the message-passing model of the WebMessaging and WebWorkers APIs.

We design formal semantics for events and message-passing to capture fundamental operations required by those APIs, and API reference implementations which are trustworthy in that they follow the respective standards and have been thoroughly tested against their official test suites. Using our formal semantics and reference implementations, we develop JaVerT.Click, the first static symbolic execution tool for JavaScript supporting both event-based and message-passing APIs.

We evaluated both the reference implementations and the symbolic execution engine of JaVerT.Click. By testing the reference implementations against their official test suites, we found coverage gaps and issues in the test suites, most of which have been since fixed. By testing the symbolic execution engine against three open-source libraries, we established the bounded correctness of functional properties and found real bugs.

‘Nothing in life is to be feared; it is only to be understood. Now is the time to understand more, so that we may fear less.’

Marie Curie

Acknowledgements

First, I would like to thank my supervisor Prof. Philippa Gardner for all her guidance and support during these four years, and for pushing me to be the best version of myself.

My second thanks goes to my co-supervisor Prof. José Fragoso Santos for his incredible dedication, friendship and work ethics. I feel privileged to have been working with him.

I would also like to thank my co-author Petar Maksimović for his technical contribution to this project.

I sincerely appreciate the financial support from CAPES Foundation during the last four years that funded my PhD project through the research grant 88881.129599/2016-01.

During this long journey, I had unconditional support from my friends at Imperial College. In special, I would like to thank Shale Xiong and Teresa Carbajo Garcia, who gave me a warm welcome when I started my PhD and made me laugh even in the most difficult moments.

I had the excellent opportunity to do internships at Amazon and Facebook during my PhD. I am very thankful to Daniel Schoepe and Franco Raimondi, who mentored me in Amazon and gave me the chance to apply my PhD project in an industrial context. I absolutely enjoyed working with them and the whole Prime Video Automated Reasoning team. During my Facebook internship, I was lucky to work with Jules Villard, who patiently guided me, providing great support and feedback in my internship project. I extend my gratitude to my amazing peers Daiva Naudžiūnienė and Ezgi Ciçeq, as well as the other members of the Infer team.

I am also grateful to my MSc supervisors Prof. Paulo Borba and Prof. Leopoldo Teixeira, who have all my admiration and respect, and kept encouraging me even after I started my PhD. I have no doubt that having been through this previous research experience made a lot of difference during the course of my PhD.

Finally, I would like to thank my family and friends for being my strong pillar and for giving me constant motivation. I am especially thankful to my husband Roberto for his friendship, loyalty, and immense support during the last 12 years; to my parents Augusto and Claudia for being my main source of inspiration and for teaching me, since I was a kid, to fight tirelessly to achieve my goals; and to my sister and best friend Débora, who makes me believe in myself and is always by my side despite our physical distance. I would not have done this without you.

Contents

List of Figures

1. Introduction

JavaScript is the *de facto* language of the Web, being widely used for the development of both client- and server-side applications [143]. These applications are a fundamental part of the Internet of today, being increasingly used to facilitate all sorts of tasks that we perform in our every-day lives, including e-mailing, text editing, online banking, and shopping. Establishing the correctness of JavaScript applications is therefore of the utmost importance, as their bugs can have severe economic and even social consequences.

The JavaScript language was created in 1995 and was soon specified in the ECMAScript standard,[1] which has significantly grown over the years. Client-side JavaScript programs execute in the browser and have access to an ever-increasing number of Web APIs. Among such APIs, the DOM [136] and HTML5 [138] are arguably the most relevant as they support the interaction between JavaScript programs and the browser as well as the HTML page displayed to the user. These APIs effectively extend the expressivity and scope of the JavaScript language by introducing, for instance, events and concurrency facilities.

The complexity of the JavaScript standard, which is now about 900 hundred pages long, renders the design of program analyses for JavaScript an extremely challenging task. When it comes to client-side JavaScript applications, this task is made even more difficult by the continuous emergence and evolution of client-side Web APIs as these APIs are implemented using a mix of different technologies and programming languages that go beyond the perimeter of JavaScript, with their source code often being neither available nor suitable for program analyses. Moreover, Web APIs commonly rely on event-driven and message-passing models, augmenting the expressiveness of the JavaScript language and therefore requiring specialised analysis techniques.

Events are at the core of client-side Web programming as they are used to model the interaction between the user and the Web page. Client-side JavaScript programs manage events with the help of various event-driven Web APIs, of which the DOM is of special importance as it provides the foundation upon which other Web APIs are built. In particular, it defines various interfaces capturing the core event-related behaviour of Web page objects, which are then used by the other APIs. Even though most Web APIs rely on the DOM event model, these APIs come with their own events and idiosyncrasies. The complexity and interconnectedness of these events complicate the control-flow structure of Web programs, rendering their analysis extremely challenging.

JavaScript programs were originally constrained to running in a single thread. Later, with the goal of improving user experience, the WHATWG group [141] introduced the WebMessaging [140] and WebWorkers [133] APIs, which extend the language with support for message-passing concurrency [19, 65, 69]. These APIs are event-driven and rely on the DOM event model, allowing for JavaScript programs to execute in a multithreaded environment with their own memories and to communicate with each other via messages. The combination of the concurrent and asynchronous natures of the

[1] `http://ecma-international.org`

WebMessaging and WebWorkers APIs adds yet another layer of complexity to the analysis of client-side JavaScript applications.

Our challenge is to design a new symbolic analysis for event-driven JavaScript applications, focussing on the *symbolic testing* of such applications. With symbolic testing, developers write tests with symbolic inputs and annotate them with assumptions and assertions, respectively expressing constraints over the given inputs and the properties that the computed outputs must satisfy. Importantly, symbolic testing can be used to establish bounded correctness of functional properties of the tested applications and also to find bugs by exploring all execution paths up to a pre-established bound [76, 126]. To this end, we implement the first static symbolic execution tool for JavaScript supporting both event-based and message-passing APIs.

Although there is a number of symbolic execution tools for analysing JavaScript programs [97, 32, 114, 33, 72, 84, 3], they either do not support any client-side Web APIs [97, 32, 114, 33], being only applicable to pure JavaScript code, or are restricted to a single event-based API [72, 84, 3]. Furthermore, to the best of our knowledge, there is no static symbolic execution tool with support for reasoning over event-based or message-passing Web applications. All the existing tools are purely dynamic and designed only for bug-finding purposes. For this reason, they do not systematically explore the program execution space and therefore cannot be used to establish bounded correctness guarantees for the analysed code.

In order to build our symbolic execution tool, JaVerT.Click, we design an *event semantics*, which captures the essence of event-based Web APIs, and a *message-passing semantics*, which models core operations of client-side message-passing APIs. Additionally, we provide JavaScript reference implementations of the DOM, JS Promises, JS async/await, WebMessaging and WebWorkers APIs. Our reference implementations faithfully model their respective Web standards and are built on top of our event semantics and message-passing semantics. We tested our reference implementations against their official test suites, making sure that they pass all applicable tests. During this process, we discovered and reported coverage gaps [52] and issues [49, 55, 48, 53, 50, 54, 51, 56] in the official test suites.

We develop JaVerT.Click on top of JaVerT 2.0 [35], our verification and testing tool for JavaScript that follows the language semantics without simplifications. We evaluate JaVerT.Click by using it to symbolically test three highly-used open-source libraries that call all the targeted APIs: `cash` [142], `p−map` [120] and `webworker-promise` [105], demonstrating that it can scale to real-world code. Importantly, this symbolic testing process allowed us to: **(1)** establish the bounded correctness of several functional properties; **(2)** find coverage gaps in the libraries' concrete test suites; and **(3)** find 6 previously unknown bugs. All discovered bugs have been reported to the corresponding development teams and have since been acknowledge and, for the most part, fixed, showing that they were found to be relevant by the developers of the corresponding projects.

1.1. The Chosen APIs

Given our choice to focus on event-driven and message-passing programming, we naturally choose Web APIs relying on either an event or message-passing model. The DOM is a natural first choice considering its popularity [94] and its underlying event model. We also consider two APIs for asynchronous programming: JS Promises and JS async/await. These two APIs are tightly connected to each other and were introduced to the JavaScript language to support the development and comprehension of asynchronous code. Importantly, JS Promises and JS async/await facilitate the structuring and organisation of JavaScript code that manages events. Hence, they are interesting targets for JaVerT.Click. Finally, we also model the WebMessaging and WebWorkers APIs due to their message-passing and event-driven nature and the nonexistence of static analysis tools to analyse them. In the following, we describe each chosen API.

DOM API. The Document Object Model (DOM) is an API through which the code executing in the browser can interact with the webpage displayed to the user. Initially designed as a simple XML/HTML inspect-update library, the DOM has been substantially extended over the last twenty years and now includes a wide variety of features, such as specialised traversals, events, abstract views, and cascading style sheets. Recently, the most relevant of these APIs, Core Levels 1-3 [130, 131], have been unified in a single all-encompassing DOM API, called the DOM Living Standard [136], which is currently being maintained by the WHATWG group [141] and defines a "platform-neutral model for events, aborting activities, and node-trees". From the DOM Living Standard, we choose to focus on: **(1)** the DOM Core Level 1, in order to provide key functionalities related to dynamic access and update of webpages; and **(2)** DOM Events, as event programming is of special interest to us.

JS Promises. Before the introduction of JS Promises, JavaScript developers would write asynchronous code through the use of *asynchronous callbacks*, which were often linked to bad programming practices. A recurring problem was the *callback hell* [37] anti-pattern consisting of the nesting of asynchronous callbacks. Additionally, due to the lack of a proper error handling mechanism when writing asynchronous callbacks, developers could informally use the *error-first convention*, which establishes that the first argument of the callback should be used for error-handling. However, developers would not necessarily use this convention. JS Promises were introduced in the 6th version of the ECMAScript (ES) standard, avoiding *callback hell* and also providing a proper error-handling mechanism.

JS async/await. The `async` and `await` operators were introduced in the 8th version of the ES standard to facilitate the development and comprehension of asynchronous code. If a JavaScript function is declared with the `async` keyword, its return value is always implicitly wrapped in a `Promise` object. The `await` operator can be used inside `async` functions and allows the current execution to be suspended until the given promise is fulfilled or rejected.

WebMessaging. To allow multiple scripts included in different windows to communicate via messages, the WHATWG group provides the WebMessaging API, which is part of the HTML5 standard [138]. The communication model defined by the WebMessaging API is an important basis for the WebWorkers API, establishing the way that workers exchange messages with one another. Messages between workers are sent asynchronously according to the message-passing paradigm [19, 65, 69]. So, if the main thread sends a message to a worker thread, the former does not block, waiting for the reply. Instead, it can still handle the user interaction and process events.

WebWorkers. Before the introduction of the WebWorkers API, JavaScript execution happened in a single thread. In order to allow scripts to run in multiple threads, the WHATWG group published the WebWorkers API as part of the HTML5 standard. Workers can handle computationally intensive tasks without blocking other scripts. This is possible because each worker runs in a separate thread, has its own memory and its own event loop. Note that both the WebMessaging and the WebWorkers

APIs rely on the DOM API, and, more specifically, on its event-related interfaces. Whenever a message arrives at a worker thread, a *message event* is triggered on that thread.

The main challenge of this thesis is the design of a new symbolic execution tool for analysing client-side JavaScript programs that interact with the DOM, JS Promises, JS async/await, WebMessaging and WebWorkers APIs. In the following, we detail our solution.

1.2. Solution

We introduce JaVerT.Click, the first static symbolic execution tool for JavaScript supporting both event-based and message-passing APIs. In contrast to existing tools supporting the analysis of event-related APIs, which focus on a particular API, JaVerT.Click can reason about JavaScript programs that use multiple event-related APIs within a unified formal semantics. Moreover, our tool enables, for the first time, static analysis of the WebMessaging and WebWorkers APIs.

In order to build our analysis, we design: **(1)** an *event semantics*, which identifies the fundamental building blocks underpinning the event models of the DOM Events, JS Promises and JS async/await APIs; **(2)** a *message-passing semantics*, which captures the essence of the message-passing model underneath the WebMessaging and WebWorkers APIs; and **(3)** trusted JavaScript reference implementations of DOM Core Level 1, DOM Events, JS Promises, JS async/await, WebMessaging and WebWorkers, the APIs that we target in this thesis. These implementations are trusted in that all except that of JS async/await follow their respective API standards line-by-line and are thoroughly tested against the official test suites, passing all the applicable[2] tests, and actually uncovering coverage gaps and issues in the official test suites. Our approach was inspired on the JSCert project [11], which proposed a formal semantics for JavaScript that closely follows the 5th version of the ECMAScript standard. The authors introduced the concept of *eyeball closeness* to establish such correspondence.

Our formal semantics for events and message-passing APIs abstract away implementation details that would be cumbersome to model at the analysis level. This is important to **(1)** guarantee that our formal semantics are compatible with multiple APIs and **(2)** avoid that they become obsolete over time given that Web standards are in constant evolution. We then leave API-specific details to our JavaScript reference implementations, which rely on our formal semantics for events and message-passing.

The key insight on the design of our formal semantics for events and message-passing is that they are fully parametric. Our event semantics is parametric on an underlying language, so that it can focus only on event-related details and filter out any clutter potentially introduced by the semantics of the language. Our message-passing semantics is designed parametrically on an underlying language semantics with support for events, as the message-passing model of client-side Web APIs relies on the DOM event model. The HTML5 standard does not define a scheduling policy to regulate the allowed interleavings between concurrent threads. Hence, browsers are free to implement the scheduling policy that they see fit. Accordingly, we make our message-passing semantics also parametric on a *scheduler* that chooses the thread to be executed at each computation step. For instance, one may use a scheduling policy that simulates the behaviour of a specific browser, or, alternatively, one that follows a highly interleaving strategy in order to detect concurrency-related bugs.

[2] We filter out tests that depend on JavaScript features not supported by our infrastructure.

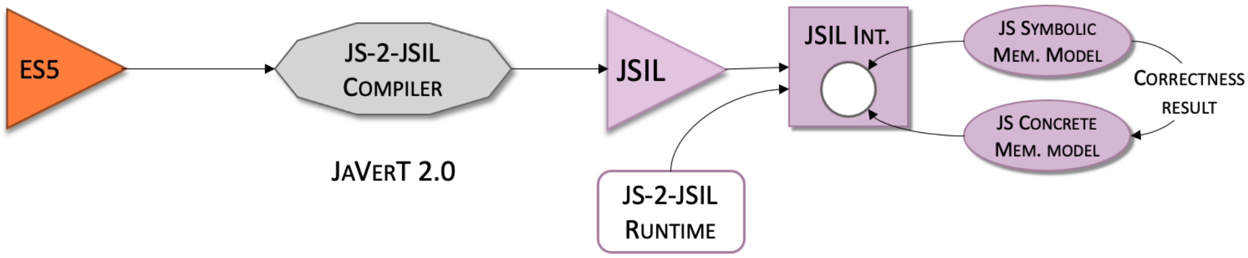

Figure 1.1.: Fragment of JaVerT 2.0 Infrastructure

We implement JaVerT.Click on top of JaVerT 2.0, our trustworthy verification and testing tool for JavaScript. In the following, we detail the infrastructure of JaVerT 2.0 and JaVerT.Click.

JaVerT 2.0 infrastructure. JaVerT 2.0 supports three kinds of analysis: **(1)** *whole-program symbolic testing*, which allows to establish the bounded correctness of functional properties [76, 127]; **(2)** *verification*, which provides full functional correctness guarantees for a given program and its specification written in separation logic [103, 18], and **(3)** *bi-abduction*, which is a compositional bug finding technique based on the automatic inference of specification summaries [16]. From the three types of analysis provided by JaVerT 2.0, symbolic testing was a natural choice for JaVerT.Click given its purpose: analysing real-world JavaScript code interacting with multiple Web APIs. We now describe the modules of JaVerT 2.0 that are related to its symbolic testing engine.

JS-2-JSIL Compiler: Compiles a JavaScript program written in ES5 Strict [27] to JSIL, the intermediate language for JavaScript of JaVerT 2.0. The JS-2-JSIL Compiler closely follows ES5 Strict and was tested against the official ES test suite;

JS-2-JSIL Runtime: JaVerT 2.0 does not rely on external JavaScript runtime; instead, it comes with its own implementation of the ES5 built-in functions and internal functions;

JSIL Interpreter: Provides a symbolic execution engine for the JSIL language. The JSIL interpreter is parametric and can be instantiated either with a concrete or a symbolic JavaScript memory model, yielding a concrete or a symbolic JSIL interpreter. JaVerT 2.0 also provides a correctness result linking symbolic and concrete JSIL executions to guarantee, for instance, the absence of false positive bug reports.

In Figure 1.1, we introduce a fragment of the infrastructure of JaVerT 2.0 that is relevant for symbolic testing purposes. Given a JavaScript program written in ES5 strict, it performs the following steps. First, the given program is compiled to JSIL using the JS-2-JSIL compiler [34, 35]. The resulting JSIL code is then executed by the JSIL interpreter, which can be instantiated with either a concrete (for testing) or a symbolic (for analysis) JavaScript (JS) memory model. JaVerT 2.0 provides a correctness result that states that its symbolic testing has no false positives.

JaVerT.Click infrastructure. JaVerT 2.0 enables the symbolic testing of JavaScript programs. Developers can write symbolic tests with the use of symbolic inputs instead of concrete ones, and include assumptions and assertions over these inputs. In order to achieve our goal, which is the analysis of client-side JavaScript programs interacting with event-driven and message-passing Web APIs, we add new modules on top of JaVerT 2.0. Figure 1.2 shows the infrastructure of JaVerT.Click, including the preexisting modules provided by JaVerT 2.0 that are relevant for symbolic testing.

Events Module: Comprises the implementation of our event semantics and is designed parametrically on an underlying language interpreter. We instantiate the Events Module of JaVerT.Click with JSIL, our intermediate language for JavaScript. Analogously to our JSIL interpreter, the Events Module can be instantiated with either a concrete or a symbolic language interpreter, and we lift the correctness result of the symbolic testing of JaVerT 2.0 up to the Events Module. If an error is reported by the Events Module symbolically, it must also happen concretely.

Message-passing Module: Comprises the implementation of our message-passing semantics and is designed parametrically on an underlying language semantics with built-in support for events and a scheduler. We instantiate our Message-passing Module with our Events Module, which is itself instantiated with the JSIL interpreter, and a scheduler that mimics the scheduling policy used by most browsers. The Message-passing Module can also be instantiated either concretely or symbolically, and comes with a correctness result which is analogous to the one provided by the Events Module.

API Reference Implementations: We also add to JaVerT 2.0 JavaScript reference implementations of our chosen APIs: DOM Core Level 1, DOM Events, JS Promises, JS async/await, WebMessaging and WebWorkers. Our reference implementations faithfully model their respective standards and were thoroughly tested against their official test suites. During the testing process, we discovered and reported [52] coverage gaps in the DOM Events official test suite. Additionally, we reported [49, 48, 50, 51] and fixed [55, 53, 54, 56] issues in the official test suites of WebMessaging and WebWorkers. The implementations of DOM Events, JS Promises and JS async/await interact with the Events Module to support core event-related features; analogously, the implementations of WebMessaging WebWorkers interact with the Message-passing Module to support core message-passing features.

ES6+ Transpiler: JaVerT 2.0 analyses JavaScript code written in ES5 strict. In order to support ES6+ features, such as the JavaScript operators `async` and `await`, we implement an ES6+ Transpiler that translates some features of ES6+ to ES5 strict.

Given a JavaScript program, JaVerT.Click performs the steps shown in Figure 1.2. First, the given program is transpiled to ES5 using the ES6+ transpiler. Then, the transpiled program, together with the reference implementations of the chosen APIs, is compiled to JSIL using the JS-2-JSIL compiler of JaVerT 2.0. The resulting JSIL code is then executed by the Message-passing Module instantiated with either a concrete or a symbolic Events Module, which is itself instantiated with the JSIL interpreter in JaVerT.Click. We also instantiate our Message-passing Module with a scheduler S that implements a scheduling policy which is analogous to the one observed in most browsers.

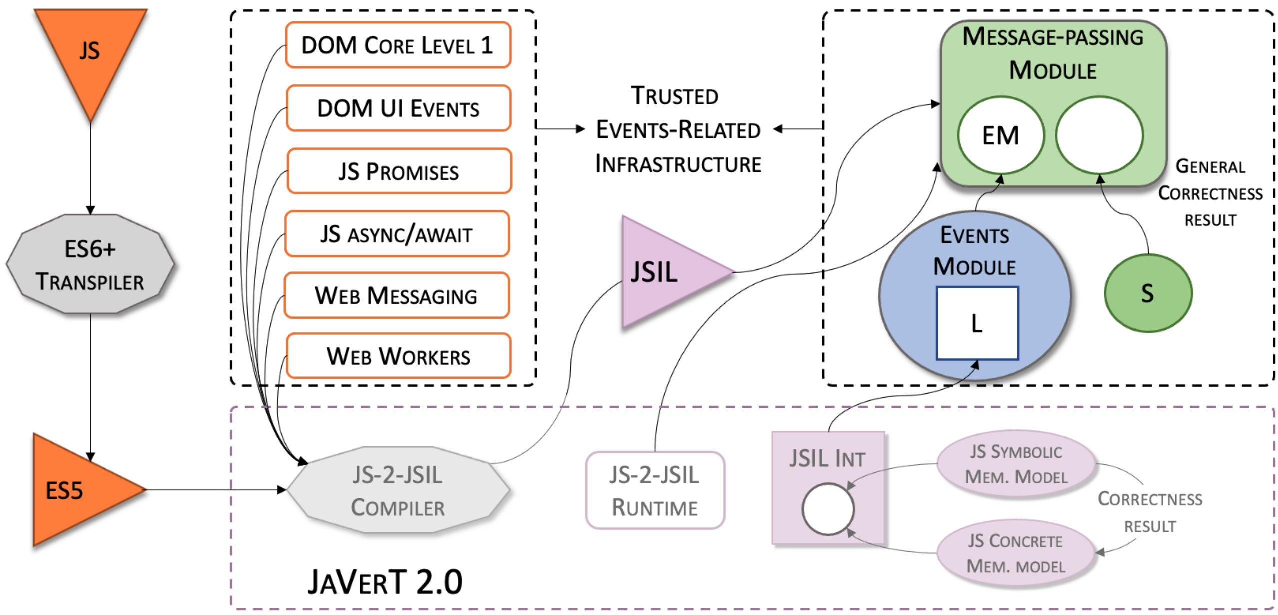

Figure 1.2.: Infrastructure of JaVerT.Click

1.3. JaVerT.Click in Practice

The evaluation of JaVerT.Click is two-fold: **(1)** we test our reference implementations of the chosen APIs against their official test suites [26, 57], guaranteeing that they pass all applicable tests, and **(2)** we use JaVerT.Click to symbolically test three real-world open-source libraries.

1.3.1. Testing Reference Implementations of the Chosen APIs

To ensure that our JavaScript reference implementations are trustworthy and follow their respective standards, we tested them against their test suites making sure that they pass all applicable tests. The number of applicable tests is 527 for the DOM Core Level 1, 56 for DOM Events, 344 for JS Promises, 68 for JS async/await, 91 for WebMessaging and 158 for WebWorkers. During this process, we discovered coverage gaps in the test suites of the DOM Core Level 1 and DOM Events APIs and wrote additional tests [52] to achieve 100% line coverage. Additionally, we found, reported [49, 48, 50, 51] and fixed [55, 53, 54, 56] four bugs in the official HTML5 test suite [57] when testing our reference implementations of the WebMessaging and WebWorkers APIs.

1.3.2. Symbolic Testing of Open-source Libraries

To evaluate the symbolic execution engine of JaVerT.Click, we symbolically tested three real-world open-source libraries: `cash` [142], `p−map` [120] and `webworker-promise` [105], which together cover all the targeted APIs. The `cash` library calls the DOM Core Level 1 and DOM Events API; the `p−map` library calls the JS Promises and JS async/await APIs; and `webworker-promise` calls both WebMessaging and WebWorkers. For each library, we wrote a symbolic test suite mostly by adapting the existing concrete test suite available in the library repository. By symbolically testing the three libraries libraries, we were able to **(1)** establish the bounded correctness of several functional properties, **(2)** achieve better coverage than concrete testing, and **(3)** uncover 6 previously unknown bugs. In the following, we summarise the symbolic testing results for `cash`, `p−map` and `webworker-promise`.

The `cash` library [142] aims at providing similar functionality to jQuery[3] while remaining as small as possible. In order to implement jQuery features, the library heavily relies on the DOM API. It has a growing community of users, with more than 21K weekly downloads on NPM[4], and more than 3M overall downloads[5], and more than 5.7K stars on GitHub [142]. By running symbolic tests for `cash` in JaVerT.Click, we established the bounded correctness of several event-related properties, achieving 100% of line coverage. Additionally, we found and reported 2 bugs to the developers [43, 42], who fixed one of them [41]. The other bug was acknowledged, but the developers argued that it would be unlikely to happen.

The `p−map` library [120] provides a tiny layer on top of JavaScript promises, allowing to apply a mapper function concurrently over an array of promises. Although being small, the `p−map` library is largely used, having more than 18M weekly downloads on NPM[6], almost 3B overall downloads[7] and 808 stars on GitHub [120]. It calls both the JS Promises and JS async/await APIs. We developed a symbolic test suite for `p−map` and were able to establish the bounded correctness of important functional properties and to achieve 98.76% of line coverage. Additionally, we found a bug in `p−map` [46], which has been fixed by the developers [44].

The `webworker-promise` library [105] is a highly-used promise-wrapper over the WebMessaging and WebWorkers APIs with 2,190 weekly downloads on NPM[8] and a total of 413,794 downloads.[9] By symbolically testing `webworker-promise`, we established the bounded correctness of several functional properties related to message-passing programming and found three bugs [59, 58, 60] and fixed two by submitting pull requests [61, 62]. The developer has agreed to fix the remaining bug.

1.4. Contributions

We design JaVerT 2.0, a symbolic execution tool for JavaScript with support for both verification and testing. JaVerT 2.0 follows the language semantics without simplifications. In order to achieve the main goal of this thesis, which is the analysis of client-side JavaScript applications calling multiple event-driven and message-passing APIs, we make the following contributions on top of JaVerT 2.0:

1. A unified event semantics that, for the first time, captures fundamental event-related operations required by the DOM Events, JS Promises and JS async/await APIs. Such operations include, for instance, event handler registration and event dispatch.

2. The first message-passing semantics that has, in its core, a message-passing model compatible with both the WebMessaging and WebWorkers APIs. The message-passing features provided include sending a message between two running threads and creating a new thread.

3. JaVerT.Click, which is the first static symbolic execution tool for JavaScript supporting both event-based and message-passing APIs. JaVerT.Click is also the first symbolic execution tool to

[3] `https://jquery.com/`
[4] `https://www.npmjs.com/package/cash-dom`
[5] `https://npm-stat.com/charts.html?package=cash-dom&from=2016-01-04&to=2022-01-04`
[6] `https://www.npmjs.com/package/p-map`
[7] `https://npm-stat.com/charts.html?package=p-map&from=2016-01-04&to=2022-01-04`
[8] `https://www.npmjs.com/package/webworker-promise`
[9] `https://npm-stat.com/charts.html?package=webworker-promise&from=2017-01-04&to=2022-01-04`

support symbolic reasoning over the WebMessaging and WebWorkers APIs.

4. Trustworthy JavaScript reference implementations of the DOM Core Level 1, DOM Events, JS Promises, JS async/await, WebMessaging and WebWorkers APIs that, except of JS async/await, follow their respective standards line-by-line and were thoroughly tested against their respective test suites. We believe that our reference implementations could be used by other static analysis tools and serve multiple purposes, such as fine-grained debugging.

5. Symbolic test suites[10] for the `cash`, `p−map` and `webworker-promise` open-source libraries, that successfully **(1)** established the bounded correctness of several functional properties, **(2)** found coverage gaps in the original concrete tests, and **(3)** uncovered 6 previously unknown bugs.

This work has been done in collaboration with my supervisors. In addition, the concrete testing of the Promises and async/await APIs, and the symbolic testing of the cash library including the extension of the first-order solver of JaVerT 2.0 with bounded string reasoning were done in collaboration with Petar Maksimović.

1.5. Thesis Outline

We detail the structure of the main body of this thesis. Chapter 2 covers the research literature that is relevant for the development of this work. We focus on formal semantics and a variety of program analysis techniques which target either any of our chosen APIs or the JavaScript language in general. Then, in Chapter 3, we introduce the JaVerT 2.0 tool, which is the technical background of this thesis. The remaining chapters are as follows:

- In Chapter 4, we introduce our event semantics. We start with a motivating example to illustrate the complexity of event-driven Web APIs. We then present our solution discussing: the parametric construction of the event semantics; our syntax for events; how the symbolic engine of JaVerT.Click can find bugs in the motivating example; and the rules for our event semantics, instantiated with either a concrete or a symbolic language semantics. Finally, we prove a correctness theorem relating the concrete and symbolic event semantics.

- In Chapter 5, we introduce our message-passing semantics. We start with a simple motivating example that was taken from the `webworker-promise` library and contains a real bug. The structure then follows analogously to Chapter 4. We present our solution discussing: the parametric construction of the message-passing semantics; our message-passing syntax; how the symbolic engine of JaVerT.Click can find bugs in the motivating example; and the rules for our message-passing semantics, instantiated with either a concrete or a symbolic language semantics with support for events. Finally, we prove a correctness theorem relating the concrete and symbolic message-passing semantics.

- In Chapter 6, we introduce our reference implementations of the chosen APIs: DOM Core Level 1, DOM Events, JS Promises, JS async/await, WebMessaging and WebWorkers. For each API, we give an overview by explaining its main interfaces, and present our reference implementation by

[10] We do not include the test suites developed for the industrial code inside Amazon because they need to remain confidential.

showing its design, the line-by-line closeness with its respective standard and how it interacts with the event semantics and the message-passing semantics.

- In Chapter 7, we discuss the evaluation of JaVerT.Click by addressing two aspects: the testing of our reference implementations against their respective test suites; and the symbolic testing of the `cash`, `p−map` and `webworker-promise` libraries. Our reference implementations pass all applicable tests from their respective official test suites and, during their testing process, we found coverage gaps and issues in the official tests. The symbolic test suites developed for the `cash`, `p−map` and `webworker-promise` libraries allows us to establish the bounded correctness of several functional properties, find coverage gaps in their concrete tests, and discover previously unknown bugs.

1.6. Publications

During the first year of my PhD, I worked on the JaVerT 2.0 tool. More concretely, I designed the formal model underpinning JaVerT 2.0 bi-abductive analysis. Afterwards, I focused on the main goal of my thesis, which is the analysis of client-side JavaScript programs that interact with multiple event-based APIs. The publications are listed below.

- José Fragoso Santos, Petar Maksimović, Gabriela Sampaio and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. POPL, 2019.
- Gabriela Sampaio, José Fragoso Santos, Petar Maksimović and Philippa Gardner. A Trusted Infrastructure for Symbolic Analysis of Event-driven Web Applications. ECOOP, 2020.
- Gabriela Sampaio, José Fragoso Santos and Philippa Gardner. Symbolic Analysis of Message-passing JavaScript Client-side Programs. *To be submitted.*

2. Related Work

A key challenge of this work is the static analysis of JavaScript programs calling Web APIs, focussing on events and message passing. In this chapter, we first give an overview of previous works tackling the design of formal semantics and analysis of Web standards (§2.1), focusing on the APIs supported by JaVerT.Click. Then, we discuss the existing formal semantics and static analysis techniques for JavaScript (§2.2).

2.1. Formal Semantics and Program Analysis for Web Standards

Most client-side Web programs rely on the ECMAScript, DOM and HTML5 standards and developers commonly introduce bugs that are caused by the misuse or misunderstanding of such standards [94]. For this reason, there are several works which focus on the design of rigorous semantics and program analyses for Web standards. To the best of our knowledge, JaVerT.Click is the first tool to support the analysis of Web programs that rely on multiple client-side Web APIs. Previous works focus on analysing programs using a specific Web standard instead of taking into consideration multiple standards. In the following, we discuss such initiatives for the APIs supported by JaVerT.Click: DOM Core Level 1 [130], DOM Events [136], JS Promises [28], JS async/await [29], WebMessaging [140] and WebWorkers [133].

Formal Semantics of DOM Core. Based on context logic [17], Smith et al. introduced an axiomatic semantics [39] for a small fragment of DOM Core Level 1, proving it sound with respect to their operational semantics. In his PhD thesis [118], Smith later extended this axiomatic semantics to all fundamental interfaces of DOM Core Level 1, including live collections and fine-grained reasoning about various types of DOM nodes, omitting only a minor part of the extended interfaces, which are: `CDATASection`, `DocumentType`, `Notation`, `Entity`, `EntityReference` and `ProcessingInstruction`. Later, Raad et al. [100] used structural separation logic [144] to give a new axiomatic semantics for a small fragment of DOM Core Level 1, improving on [39] and [118] by allowing for more compositional reasoning about DOM clients. This axiomatic semantics follows the DOM standard closely, but has not been implemented, and there has been no work on using this semantics to reason about real-world JavaScript programs that interact with the DOM; only pen-and-paper proofs for small examples are provided.

Brucker et al. defined an extensible and executable formal semantics [13] for the Core DOM 4[1] in Isabelle/HOL. Their model allows to prove properties about the DOM, for example: *"Inserting a node in the DOM tree using* `insert_before` *never leads to duplicates in the node's children list."*. The theory covers the following core datatypes of the DOM specification: `Document`, `Node`, `Element` and `CharacterData`, as well as a subset of the methods exposed by these interfaces, such as

[1] DOM Level 4: `https://www.w3.org/TR/2015/WD-dom-20150428/`

get_element_by_id and create_element. Other types of nodes, such as Text and EntityReference, and event handling features defined by the EventTarget interface are not supported. Although the formalisation is executable, it was not tested against the official test suite. In contrast, our DOM reference implementation follows the standard line-by-line and passes all applicable tests of the official test suite.

Several operational semantics for different fragments and adaptations of DOM Core Level 1 were proposed for various types of analyses, such as information flow control [92, 106], type systems [124] and abstract interpretation [73], targeting JS programs that interact with the DOM. These papers, however, do not aim to establish a trusted formal representation of the DOM using which others can build their own program analyses; instead, they provide a DOM representation specific to their kind of analyses. In contrast, our DOM Core Level 1 JS reference implementation has been designed to be trusted in that it follows the text of the standard line-by-line and passes all tests of the official test suite [129]. This, combined with its extensive use in the symbolic testing of the cash library, gives us confidence that others will be able to use it for their analysis of JS programs calling the DOM.

Symbolic Analyses for the DOM Core. Symbolic reasoning about the DOM in the literature is mostly focussed on bug-finding and/or automatic concrete test generation. For example, ConFix [30] uses concolic execution to automatically generate DOM fixtures[2] that allow high-coverage testing of JavaScript functions that use the DOM. More concretely, ConFix deduces DOM-dependent path constraints based on a given trace of the program under test, and then obtains a DOM-based fixture by calling a XML constraint solver with the deduced constraints. Although the approach has shown to increase coverage of DOM-dependent functions, it does not support DOM events and dynamically generated code using eval that interacts with the DOM. It is not our goal with JaVerT.Click to generate DOM fixtures; instead, we aim at using static symbolic execution to prove the bounded correctness of functional properties. Furthermore, our DOM reference implementation supports both DOM Core Level 1 and DOM Events.

Zou et al. introduced the notion of Virtual DOM (V-DOM) coverage [147], a novel criterion for testing dynamic Web applications which aims at increasing code coverage. By using control-flow and data-flow analysis techniques, the tool computes a virtual DOM tree, which consists of a logical representation of all possible DOM trees that could be generated by different execution paths of server scripts. The V-DOM construction enables a more effective testing of Web applications calling the DOM. One of the limitations of this approach is that although it can support DOM events, it only considers handlers that were registered statically (via HTML code, e.g. `<button onclick="myFunction()"/>`), meaning that it does not support dynamic handler registration (via the DOM function addEventListener()). JaVerT.Click aims at analysing client-side code and relies on a semantics for events that captures the essence of multiple Web event-driven APIs. Additionally, our analysis supports dynamic event handler registration/deregistration and event dispatching.

When it comes to research on securing client-side Web applications, various symbolic analyses have been proposed with the goal of finding DOM-based XSS vulnerabilities [82, 95, 110]. This type of vulnerability was first described by Amit Klein in 2005 [78] and has affected big tech companies such

[2] A test fixture refers to the fixed state required for testing a system. For instance, ConFix uses DOM trees as test fixtures.

as Google[3], Yahoo[4] and Twitter.[5] A DOM-based XSS attack is caused by unsafe data flows during the DOM manipulation in client-side code. For instance, one could have the value of `document.url`, which is susceptible to attacks, being printed to the webpage through the use of the DOM function `document.write`(). In order to find such attacks automatically, Saxena et al. [110] developed FLAX, a framework which finds DOM-based XSS attacks by using taint enhanced blackbox fuzzing [63], a hybrid approach that combines fuzz testing and taint analysis. Then, Lekies et al. [82] adapted a browser engine to achieve support for dynamic taint-tracking. Later, Parameshwaran et al. [95] introduced DexterJS, a browser-independent tool which also applies taint tracking to find and validate DOM-based XSS vulnerabilities. All three tools have successfully found DOM-based XSS vulnerabilities in real-world programs. The purpose of JaVerT.Click, in contrast to these tools, is not to find security vulnerabilities, but to verify functional properties related to the use of several Web APIs. We believe that the symbolic execution engine at the core of JaVerT.Click can be adapted to find DOM injection attacks by instrumenting with a security monitor in the style of [111].

Formal Semantics of DOM Events. There are very few formal semantics for DOM Events. In this context, the work closest to ours is [83], which presents the first operational model for reasoning about DOM events. This model consists of a Scheme[6] reference implementation of DOM UI events and is used to prove meta-properties of the DOM semantics, such as the immutability of the propagation path during the execution of the `Dispatch` algorithm. The authors justify their reference implementation by annotating the paragraphs of the standard with links to the relevant definitions and reduction rules in their implementation, and by comparing its behaviour with various browser implementations using randomly generated test cases. The implementation, in contrast to JaVerT.Click, is not tested against the official DOM Events test suite and does not have a line-by-line correspondence with the text of the DOM standard.

Rajani et al. [101] propose a simplified DOM event semantics instrumented with a sound information-flow monitor, and implement the monitor instrumentation on top of Webkit,[7] the browser engine used by Safari. The proposed semantics is, however, only intended for illustrative purposes as it does not include a number of event-related features, such as interaction with shadow trees, slotables, and touch/related targets. In contrast, our reference implementation of DOM Events does not simplify the standard and passes 100% of the appropriate tests, given our current coverage (56 tests in total).

Static and Dynamic Analyses for DOM Events We discuss several program analysis tools for DOM events grouping them into three categories: **(1)** tools based on static analysis [96, 72, 123], **(2)** tools based on dynamic analysis [135, 4], and **(3)** tools that combine the advantages of static and dynamic analysis [3, 98, 84].

In 2011, Jensen et al. [72] introduced the first static analysis tool that provide a detailed model of the DOM on top of TAJS [74, 75]. The goal was to identify programming errors such as unreachable code and function calls with the wrong number of arguments. Their DOM model is not complete, only supporting the DOM features that the authors believe are mostly used. Additionally, it only

[3] `https://blog.mindedsecurity.com/2012/11/dom-xss-on-google-plus-one-button.html`
[4] `https://www.exploit-db.com/docs/english/24109-domsday---analyzing-a-dom-based-xss-in-yahoo!.pdf`
[5] `https://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-something.html`
[6] `https://schemers.org/Documents/Standards/R5RS/r5rs.pdf`
[7] `https://webkit.org`

supports ES3. Then, Park et al. introduced $\text{SAFE}_{\text{WApp}}$ [96], a static analysis framework built on top of SAFE [81] to find bugs in client-side Web programs. The authors performed an empirical study in order to guide the modeling process of the DOM. Because not all DOM features are supported, the analysis can be unsound and imprecise. JSDEP [123] implements the first constraint-based declarative program analysis for computing dependencies between event handlers. This approach is shown to be effective, but no soundness guarantees are provided. JaVerT.Click, despite being a static analysis tool, provides a complete and trustworthy reference implementation of the DOM, which has been tested against the official test suite and could be reused in different settings. Moreover, the symbolic execution engine of JaVerT.Click can prove the bounded correctness of functional properties.

SAHAND [4] and EVMIN [135] use dynamic analysis for collecting traces of event-driven JavaScript applications with different purposes. SAHAND is a browser-independent tool that computes a temporal and context-sensitive model of the JavaScript execution, including both the client and server sides, in order to facilitate the understanding of JavaScript asynchronous code. EVMIN tries to find the minimal event traces to reproduce a given failure. This is achieved by removing events from the trace that do not affect the failure. However, their DOM model is not fully precise and it can generate infeasible event traces. JaVerT.Click, in contrast, does not compute event traces but provides a complete DOM model. Additionally, because JaVerT.Click relies on static symbolic execution, it can have more scalability issues than tools based on dynamic approaches such as SAHAND and EVMIN. On the other hand, JaVerT.Click does not give false positive bug reports.

Finally, there are tools which combine static and dynamic analysis techniques to compute event-handler dependencies, such as SYMJS [84], TOCHAL [3] and EHA [98]. SYMJS [84] performs a dynamic write-read analysis and uses this information to automatically generate tests in the form of event triggering sequences. However, its representation of the DOM is not entirely consistent with the standard: e.g., text inputs and radio boxes are represented symbolically either as strings or numbers, rather than objects. TOCHAL [3] provides a hybrid DOM-sensitive change impact analysis. Its hybrid model aims at combining the benefits of static and dynamic analysis. Although the tool seems to be effective in supporting developers on their tasks, it is not necessarily sound. EHA [98] is a bug-finding framework and introduces a novel mechanism which, in contrast to whole-program analysis, deals with *partial* execution flows triggered by events. EHA is parametric on a generator of event sequences and a static analyser, and it has been instantiated with a manual event generation and the SAFE analyser. It also does not provide soundness guarantees.

While the goals of these tools are different from ours, there is room for comparison. In particular, some of them do not follow the DOM standard (e.g., SYMJS relies on HTMLUnit [40], which provides its own implementation of the DOM event dispatch algorithm) and do not provide a proper justification with respect to their representation of the DOM. In contrast, we provide trustworthy reference implementations of DOM Core Level 1 and UI Events that follow the standard line-by-line and pass all of the applicable official tests. Importantly, these tools do not appear to be able to reason about events whose *type* is symbolic. We believe that this is one of the advantages of our work, as it allows us to write few symbolic tests to achieve code coverage. It also enables us to provide bounded correctness guarantees of library properties, which, to our knowledge, has not been done before, and which is certainly beyond the reach of either manually- or automatically-generated concrete test suites. On the other hand, JaVerT.Click does not generate tests automatically - the developers have to write symbolic tests themselves.

Formal Semantics of JS Promises. Madsen et al. [87] were the first to propose a formal core calculus for reasoning about JavaScript (JS) promises. Concretely, they introduce λ_p, an extension of the small core JavaScript calculus, λ_{JS} [68], with dedicated syntactic constructs for promise creation and manipulation. The authors give the formal semantics of λ_p and show how it can be used to encode promise operations not directly supported in the syntax (e.g. `catch` and `then`). The paper further introduces the concept of *promise graphs*, a program artifact used by the authors to explain promise-related errors.

Loring et al. [86] proposed λ_{async}, another extension of λ_{JS} that models the semantics of JS Promises, as well as NodeJS event loops. The authors define a *Priority Semantics* for λ_{async} which guarantees that different groups of asynchronous computations have different priorities. For instance, that callbacks scheduled for the `setTimeout` and `setInterval` have a different priority of I/O callbacks. One could use different scheduling policies based on the notion of priority. The authors do not guarantee, however, that the proposed schedulers strictly follow the NodeJS event loop specification.[8]

Later, Alimadadi et al. [5] extended promise graphs to take into account previously unmodelled ES6 features, such as default reactions, exceptions, `race` and `all`. Using the extended promise graphs, the authors develop PromiseKeeper, a dynamic analysis tool built on top of Jalangi [115] for finding and explaining promise-related bugs in JS code.

While the λ_{JS}-calculus [68] was justified by a desugaring function from ES5 that was tested against the official Test262 test suite [26], λ_p and λ_{async} do not come with a desugaring function from ES6 to λ_p and hence have not been tested against the promises-related part of Test262. In contrast to λ_p, which is mainly used to explain buggy behaviours related to the misuse of JS promises, our goal was to create a trusted reference implementation of JS promises that models their semantics precisely in order to enable various types of analysis for JS programs that use promises, including symbolic testing. For this reason, we took great care in justifying its correctness.

Static Analyses for JS Promises and JS async/await. Most of the static analysis tools [12, 36, 9, 66] in this domain perform program transformations to introduce or improve the use of JS Promises and JS async/await.

Before the introduction of JS Promises in ES6, asynchronous programming in JavaScript was possible through the use of asynchronous callbacks, which are commonly associated to bad programming practices, such as *callback hell*[9] and the *error-first*[10] convention. JS Promises are an alternative to asynchronous callbacks and help to deal both with the callback nesting and error-handling problems. Brodu et al. [12] and Gallaba et al [36] proposed different mechanisms to facilitate the migration of asynchronous callbacks to JS Promises. The compiler introduced by [12] transforms nested callbacks into a sequence of *Dues*, which are a simplified version of promises. PromiseLand [36] can automatically detect asynchronous callbacks and refactor them into promises.

After the introduction of the `async` and `await` operators in ES8, refactoring tools emerged to support the handling of IO-related operations. Arteca et al. [9] developed *ReSynchronizer*, a tool for

[8] `https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/`

[9] Callback hell is an anti-pattern which consists of the nesting of complex callbacks.

[10] According to the error-first convention, the first argument of the callback is reserved for error-handling and the other parameters for passing data. Although this is a widely known convention, developers may not use it.

static interprocedural side-effect analysis that increases asynchrony in JavaScript programs, consequently improving their performance. The tool automatically detects IO-related `await` expressions that causes *oversynchronisation*, and refactors such expressions so that the IO-operation is computed asynchronously as early as possible and the awaiting for that operation happens as late as possible. Gokhale et al. [66] developed *Desynchronizer*, a JavaScript refactoring tool that replaces synchronous calls to IO-related APIs with asynchronous calls through the use of the `async` and `await` operators. Both tools do not provide behaviour preservation guarantees. It is the job of the developer to make sure that behaviour is preserved after each transformation.

All the aforementioned tools have different goals than JaVerT.Click. While their focus is to perform automatic code transformations to increase asynchrony and improve performance, JaVerT.Click provides a trusted infrastructure for reasoning about client-side JavaScript programs calling multiple Web APIs. Our infrastructure includes **(1)** formal models for events and message-passing that capture the essence of the DOM, JS Promises, JS async/await, WebMessaging and WebWorkers APIs and **(2)** trustworthy reference implementations of the targeted APIs that can be used for various purposes.

Formal Semantics and Analysis tools for WebMessaging and WebWorkers. To the best of our knowledge, there is no previous work that provides a formal semantics for WebMessaging and WebWorkers yet. Hence, our message-passing semantics is the first formal model for these two Web APIs. Furthermore, we were not able to find analysis tools targeting the WebWorkers API. There are a few tools [119, 121], however, to analyse programs calling the WebMessaging API that are focused on finding security vulnerabilities in `onmessage` handlers. Since the `postMessage` function defined by the WebMessaging API allows for cross-origin communication, `onmessage` handlers must check the integrity and confidentiality of messages. In fact, according to the results obtained by Son and Shmatikov [119], there is a substantial amount of insecure handlers. The authors proposed the RvScope framework, which is a Chrome extension that collects information about `onmessage` handlers via code injection to detect potential vulnerabilities. Later, Steffens and Stock [121] developed PMForce, a dynamic execution framework based on forced execution [77] and taint tracking that reports vulnerabilities in `onmessage` handlers.

In contrast to JaVerT.Click, both these tools run in the browser engine and are not based on a formal model of the WebMessaging API. Moreover, they do not provide reference implementations of WebMessaging and do not reason symbolically about web-messaging APIs. As discussed in §6.5.2, JaVerT.Click does not support the cross-origin communication feature and it is not our goal to find security violations in `onmessage` handlers. However, JaVerT.Click comes with reference implementations of WebMessaging and WebWorkers that can be used for several purposes, such as fine-grained debugging. Moreover, JaVerT.Click introduces the first symbolic execution engine targeting the WebMessaging and WebWorkers APIs, being able to establish the bounded correctness of functional properties and find bugs in real-world code.

2.2. Formal Semantics and Program Analyses for JavaScript

In the following, we describe initiatives on the design of rigorous semantics and analysis tools for JavaScript.

Formal Semantics for JavaScript. The first formal specification of the JavaScript language was proposed by Maffeis et al. [89] and closely follows the 3rd version[11] of the ECMAScript standard. The authors defined a small step operational semantics which is not mechanised yet. The authors also report inconsistencies between existing implementations and the standard.[12]

Two years later, Guha et al. [68] introduced λ_{JS}, a core language that captures essential features of the 3rd version of the ECMAScript standard, such as prototype inheritance and the `instanceof` operator. However, λ_{JS} does not support the `eval` function, which allows to dynamically evaluate JavaScript code given as a string argument. To validate the semantics, the authors mechanise λ_{JS} using PLT Redex [31]. Politz et al. [99] build a semantics named S5 on top of λ_{JS} by including features of the ES5 strict standard, such as *getters* and *setters*, and also the `eval` function. S5 was tested against the official Test262 test suite from the ECMAScript standard.

Bodin et al. [11] then introduced JSCert,[13] a pretty-big-step semantics of ES5 which was mechanised in the Coq[14] proof assistant. JSCert was the first project that emphasised the importance of having formal semantics closely following Web standards - what the authors named *eyeball closeness*. This is a well accepted methodology to establish trust in reference implementations. In fact, we follow the same methodology in JaVerT.Click. The authors introduce the term to indicate that JSCert strictly follows the ES5 standard. The authors also provide JSRef, an executable reference interpreter implemented in Coq and extracted to OCaml, and give a soundness proof that JSRef programs have the behaviour specified by JSCert. Additionally, JSRef was tested against the Test262 test suite. In this process, the authors found bugs in all browser implementations.

JSCert was later extended by Gardner et al. [38] to include the Google's V8 Array library.[15] During this process, the authors improved the JSRef interpreter and fixed errors related to the interpretation of the ES5 standard both in JSCert and JSRef. Additionally, a more detailed analysis of the testing process was provided.

Park et al. [97] developed KJS by formalising the ES5 standard as a small-step semantics in the K framework [104]. KJS was tested against the test suite of ES5, passing all 2,782 tests. The authors also assessed the coverage of the official ES5 test suite, finding uncovered semantic rules. The additional tests uncovered bugs in industrial reference implementations, such as V8 and Safari. One of the advantages of KJS with respect to other formal semantics is that it also provides a symbolic execution engine for JavaScript programs.

Finally, Charguéraud et al. [20] proposed JSExplain, which provides a formal semantics for JavaScript using a tailor-made language to ease readability for developers. JSExplain also comes with an interactive debugger that supports the step-by-step execution of the ECMAScript specification given a JavaScript program. Although JaVerT.Click does not include an interactive debugger, its reference implementations of the various Web APIs could also be used for debugging purposes.

All these aforementioned approaches do not support formal reasoning about event-driven/message-passing Web APIs like JaVerT.Click. Our tool is built on top of JaVerT [34, 35], which provides a formal semantics for JSIL (its intermediate language for JavaScript) and has been thoroughly tested against the ECMAScript5 official test suite. Additionally, all reference implementations provided by

[11] `https://www.ecma-international.org/wp-content/uploads/ECMA-327_1st_edition_june_2001.pdf`
[12] `http://jssec.net`
[13] `http://jscert.org`
[14] `https://coq.inria.fr/`
[15] `https://v8.dev`

JaVerT.Click have been thoroughly tested against their respective official test suites.

Symbolic Execution for JavaScript. As JavaScript Web programs commonly call external APIs/libraries, there are a few general-purpose symbolic execution tools for JavaScript. Kudzu [109] was the first framework to implement symbolic execution for client-side JavaScript code. The tool comes with an automatic test generator that aims at obtaining high coverage, but it is limited to finding code injection vulnerabilities. Jalangi [114] is a framework that enables the development of dynamic analyses and combines the record-replay and shadow execution techniques to find errors due to, for instance, the presence of JS `null` or `undefined`. The use of shadow values and shadow execution allows, for instance, the development of a symbolic execution technique. For scalability reasons, the tool does not follow the semantics of the language precisely.

Later, at least two other approaches [23, 7] were developed using Jalangi. Dhok et al. [23] added type awareness to the concolic testing engine of Jalangi in order to reduce the number of generated inputs for JavaScript programs. More concretely, the approach is inter-procedural and includes a type inference mechanism. The results showed that only 5% of the original inputs generated by Jalangi were needed. Amadini et al. [7] applied constraint programming (CP) techniques to Jalangi, obtaining the Aratha tool. Instead of using standard SMT solvers, Aratha uses the CP solver G-Strings [8], which is an extension of Gecode.[16] Results showed that CP-based approaches can compete with SMT. The authors do not guarantee correctness or completeness of their JavaScript model with respect to the ECMAScript standard.

There are also symbolic execution and static analysis tools [88, 122] for JavaScript targeting NodeJS applications. Madsen et al. [88] developed Radar, a static analysis tool for event-driven NodeJS applications to detect event-related bugs, such as *dead events*.[17] The tool computes an *event-based call graph*, which is a variation of the standard call-graph to handle event-driven flow of control in JavaScript applications. The authors defined λ_ϵ, a core calculus for event-driven JavaScript applications which extends λ_{JS} with new operators such as `listen` and `emit` in order to respectively model event handler registration and event dispatch. These event-based operators are also supported by JaVerT.Click. However, we focus on establishing bounded correctness of functional properties. Additionally, while Radar was designed for NodeJS applications, JaVerT.Click primarily aimed at client-side applications.

Sun et al. [122] developed NodeProf on top of Graal.js [145], a dynamic analysis framework for JavaScript targeting NodeJS applications. One of the key ideas behind NodeProf is the possibility to activate or deactivate an analysis with zero overhead when no analysis is active. The authors claim that NodeProf can be up to three orders of magnitude faster than its main competitor, Jalangi.

In contrast to most of these tools, which perform dynamic analysis and do not model the behaviour of Web APIs, JaVerT.Click relies on static symbolic execution and models the DOM, JS Promises, JS async/await, WebMessaging and WebWorkers APIs. JaVerT.Click is built on top of JaVerT [34, 35], which faithfully models the strict version of the ES5 standard. Recently, the JaVerT tool evolved to Gillian [32, 91] a multi-language platform which was instantiated with JavaScript and C. We intend to integrate JaVerT.Click and Gillian in future by adding, on top of the framework, our semantics for event-driven and message-passing Web programs, as well as our reference implementations of all APIs

[16]Gecode: Generic constraint development environment; `http://gecode.org`.

[17]An event is dead when it triggers no handlers.

supported by JaVerT.Click. This could eventually allow us to instantiate both our event semantics and message-passing semantics with further programming languages.

Finally, concolic execution has been applied to other languages in addition to JavaScript. For instance, DART [64, 112] provides concolic execution for C programs. The tool provides a static analysis component, which can automatically extract an interface for a given program, and a dynamic analysis component, which aims at automatic test generation using random testing. CUTE and jCUTE [113] implement concolic execution techniques for C and Java to reduce redundant test cases and false warnings. JaVerT.Click, in contrast, is a static analysis tool. Static analysis techniques may not be the best when it comes to scalability, but they cover all possible execution paths. Given the purpose of JaVerT.Click, we believe that static symbolic execution is the most appropriate technique.

Dynamic Analyses for Detecting Event Races. Although JavaScript event-driven applications often run in a single thread, they are susceptible to concurrency errors typically found in multithreaded applications. Client-side JavaScript programs allow for event handlers to run non-deterministically, potentially leading to event races. In order to detect such event races, one needs to explore multiple scheduling policies of event handlers. There are a few techniques for identifying event races based on different types of dynamic analyses [102, 71, 2].

Raychev et al [102] defined the notion of *race coverage* to avoid reporting races that would never occur, consequently reducing significantly the number of false positives. The authors proposed a new dynamic analysis algorithm based on vector clocks [117] that computes races in event-driven programs. The algorithm was implemented in the Event Racer tool, which exposed 75 harmful races in a public web sites.

Later, Jensen et al [71] developed a new stateless model checker, R^4, which is based in Partial Order Reduction [6]. The technique allows for eliminating paths during the analysis that lead to the same state, consequently reducing redundant sequences of event handlers. The authors applied R^4 to 32 popular websites, in which it reported 275 warnings about potentially harmful races.

Finally, Adamsen et al [2] developed a technique based on dynamic analysis that combines approximate [70] and adverse [1] execution. The technique was implemented in a tool, InitRacer, that aims at three types of event races: form input overwritten, late event handler registration and access before definition. Empirical results showed that InitRacer could identify 1085 event races on 100 popular websites.

All the aforementioned tools are solely focussed on detecting event races. JaVerT.Click has a different goal: finding native JavaScript errors in client-side event-driven applications. However, we could adapt JaVerT.Click to also detect event races. This would require exploring multiple scheduling policies of event handlers and threads and is part of our future work.

3. An Overview of JaVerT 2.0

In order to achieve our main goal, which is the analysis of client-side Web programs calling multiple event-driven APIs, we design a new symbolic execution tool, JaVerT.Click. This tool is built on top of JaVerT 2.0 [35], our verification and testing tool for JavaScript which follows the language semantics without simplifications. JaVerT 2.0 supports three types of analysis: whole-program symbolic testing [76, 127], verification based on separation logic (SL) [103, 18], and automatic compositional testing based on bi-abduction [16]. In this thesis, we focus *only* on the symbolic testing aspect of JaVerT 2.0.

Symbolic testing was the natural first choice. While verification allows us to prove that a program satisfies a given specification, it requires writing pre- and post-conditions. In the case of JaVerT 2.0, developers need to be experts in SL in order to write such specifications. This becomes critical with the use of multiple event-driven APIs such as the DOM [136], as verifying a program would require writing specifications for the entire DOM structure of a webpage. Bi-abduction [16] is a compositional program analysis technique which tries to automatically infer a program specification in order to find bugs. Bi-abduction is less scalable than standard symbolic execution techniques given that it has to explore a much greater number of possible execution paths. In the following, we explain the usage of JaVerT 2.0 by giving a motivating example (§3.1), and introduce its symbolic execution mechanism (§3.2), which is relevant for symbolic testing testing purposes. This chapter provides an overview of JaVerT 2.0. More details can be found in [35].

3.1. Using JaVerT 2.0 for Symbolic Testing

JaVerT 2.0 is a tool for JavaScript developers, designed to assist them in the testing and verification of their programs. It is not meant for analysing JavaScript code in the wild. Although JaVerT 2.0 supports three kinds of analysis, we demonstrate only how JaVerT 2.0 can be used by developers for whole-program symbolic testing which is the type of analysis supported by JaVerT.Click.

In this section, we illustrate the usage of JaVerT 2.0 using the example of an *expression evaluator*, given in Figure 3.1 (left). It contains three functions: `evalExpr`, for evaluating a given expression under a given store; `evalUnop`, for applying a given unary operator to a given value; and `evalBinop`, for applying a given binary operator to two given values. Expressions are evaluated with respect to a store, which is assumed to be a key-value map exposing a method `get` for recovering the value associated with a given key (we re-use the key-value map implementation of [108]). For brevity, we omit the store initialisation in the symbolic tests of Fig. 3.1 (right). Expressions are represented in memory as AST objects. For instance, the expression `x + 1` corresponds to the object:
`{ type: "binop", op: "+", left: { type: "var", name: "x"}, right: { type: "lit", val: 1} }`.

Developers commonly write unit tests to check that, given some concrete inputs, their code produces the desired outputs. With JaVerT 2.0, developers can write unit tests with *symbolic* inputs and outputs, and use simple assertions to describe the properties that the outputs must satisfy. JaVerT 2.0

```
1  function evalExpr (store, e) {
2    if (typeof e !== "object") throw new Error ("E:Type");
3    switch (e.type) {
4      case "lit"    : return e.val
5      case "var"    : return store.get(e.name)
6      case "unop"   :
7        var arg_v = evalExpr(store, e.arg);
8        return evalUnop (e.op, arg_v)
9      case "binop" :
10       var left_v  = evalExpr(store, e.left);
11       var right_v = evalExpr(store, e.right);
12       return evalBinop (e.op, left_v, right_v)
13     default : throw new Error("Expr") }
14 }
15
16 function evalUnop (op, v) {
17   switch (op) {
18     case "-"   : return -v
19     case "not" : return !v
20     case "abs" : return v < 0 ? -v : v
21     default    : throw new Error ("UnOp") }
22 }
23
24 function evalBinop (op, v1, v2) {
25   switch (op) {
26     case "+"   : return v1 + v2
27     case "-"   : return v1 - v2
28     case "or"  : return v1 || v2
29     case "and" : return v1 && v2
30     default    : throw new Error("BinOp") }
31 }
```

Symbolic test 1:

```
1 var x = symb();
2 assume(typeof x !== "object");
3 try { evalExpr(store, x) } catch (e) {
4   assert(e.message === "E:Type")
5 }
```

Symbolic test 2:

```
1 var x = symb();
2 assume(typeof x === "object");
3 try { evalExpr(store, x) } catch (e) {
4   var msg = e.message;
5   assert (msg === "Expr" || msg === "UnOp" ||
6           msg === "BinOp");
7 }
```

Symbolic test 3:

```
1  var n = symb_number(), op = symb_string();
2  var lit = { type: "lit", val: n };
3  var e   = { type: "unop", op: op; arg: lit };
4  assume (op !== "not");
5  try {
6    var ret = evalExpr(store, e);
7    var abs_ret = n < 0 ? -n : n;
8    assert (((op === "-")   && (ret === -n)) ||
9            ((op === "abs") && (ret === abs_ret)));
10 } catch (e) {
11   assert(e.message === "UnOp")
12 }
```

Figure 3.1.: Expression Evaluator implementation (left); symbolic tests (right)

allows users to symbolically execute such tests, providing *concrete counter models* in case of failure which the developer can potentially use to correct the error.

Symbolic tests are more effective than concrete ones, as one single symbolic test often covers a range of program executions, each corresponding to a single concrete unit test. For instance, consider Symbolic test 1 in Figure 3.1 (right), which tests the behaviour of `evalExpr` on all non-object inputs. We would need five concrete tests to perform the same check (corresponding to the cases when the type of the input is equal to `"undefined"`, `"boolean"`, `"number"`, `"string"`, or `"function"`).

Despite its simplicity, the code of the expression evaluator exposes a common JavaScript bug. To understand the bug, consider Symbolic test 2, which states that, when given an input of type `"object"`, `evalExpr` does not throw a JavaScript native error. This is tested in lines 7-9, by asserting that if an error is thrown, it must correspond to one of the errors thrown by the code itself. However, running JaVerT 2.0 on this test returns a concrete counter-example for the input: `e` = `null`. Indeed, if we run `evalExpr` with `e` set to `null`, the JavaScript semantics throws a *type error*. This happens because, in JavaScript, `typeof null` evaluates to `"object"`, meaning that the execution reaches line 3 of `evalExpr`, causing the program to throw an error when trying to access the property `"type"` of `null`.

Finally, Symbolic test 3 covers three different behaviours of the `evalExpr` and `evalUnop` functions at the same time, effectively testing all possible behaviours of `evalUnop` on numeric inputs.

3.2. JSIL Symbolic Execution

We give an overview of the symbolic execution engine at the core JaVerT 2.0. We first give the syntax of JSIL and comment on its expressivity (§3.2.1) and then describe its symbolic semantics (3.2.2).

3.2.1. JSIL Syntax

JSIL is a simple goto language with top-level procedures and commands that operate on object heaps. Importantly, JSIL retains the dynamic features of JavaScript: extensible objects and dynamic code evaluation. In JavaScript, dynamic code evaluation is possible with the use of the `eval` function, which allows to dynamically execute commands given as a string, including object creation and function calls. The JSIL language allows for extensible objects and provides a mechanism for dynamic code evaluation, detailed shortly. In Figure 3.2, we give the JSIL syntax.

$$\lambda \in \mathcal{L}it ::= n \in \mathcal{N} \mid b \in \mathcal{B} \mid s \in \mathcal{S} \mid \mathsf{undefined} \mid \mathsf{null} \mid \mathsf{empty} \mid l \in \mathcal{L} \mid \tau \in \mathcal{T} \mid f \in \mathcal{F}id \mid \overline{\lambda} \mid \{\overline{\lambda}\} \qquad e \in \mathcal{E}xp ::= \lambda \mid x \in \mathcal{X} \mid \hat{x} \in \hat{\mathcal{X}} \mid \ominus\, e \mid e \oplus e$$

$$bc \in \mathcal{B}cmd ::= \mathsf{skip} \mid x := e \mid x := \mathsf{new}\,(e) \mid x := [e, e] \mid [e, e] := e \mid \mathsf{delete}\,(e, e) \mid x := \mathsf{hasProp}\,(e, e) \mid x := \mathsf{getProps}\,(e) \mid x := \mathsf{metaData}\,(e)$$

$$c \in \mathcal{C}md ::= bc \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [e]\ i,\ j \mid \mathsf{assume}\,(e) \mid \mathsf{assert}\,(e) \mid x := \mathsf{arguments} \mid \mathsf{return} \mid \mathsf{throw} \mid x := e(\overline{e})\ \mathsf{with}\ j \mid x := \mathsf{extern}\ e(\overline{e})\ \mathsf{with}\ j \mid x := \mathsf{apply}\,(e, e)\ \mathsf{with}\ j$$

$$proc \in \mathtt{Proc} ::= \mathsf{proc}\, f(\overline{x})\{\overline{c}\} \qquad p \in \mathcal{P} ::= \{\overline{proc}\}$$

Figure 3.2.: JSIL Syntax

JSIL *literals*, $\lambda \in \mathcal{L}it$, include numbers, booleans, strings, the special values undefined, null, and empty, object locations, types, procedure identifiers, and lists and sets of literals. JSIL *expressions*, $e \in \mathcal{E}xp$, include literals, program variables x, symbolic variables $\hat{x}$, and various unary and binary operators.

JSIL *basic commands*, $bc \in \mathcal{B}cmd$, are used for the manipulation of extensible objects and have no impact on the control flow of the program. They include: the skip command; variable assignment; object creation; property access, assignment, deletion, membership, and collection; and object metadata collection.

JSIL *commands*, $c \in \mathcal{C}md$, include basic commands, conditional and unconditional gotos, procedure calls, commands for assuming and asserting facts about the execution of the program, and new commands for argument collection, external procedure calls, procedure application, and procedure termination. The unconditional goto $\mathsf{goto}\ i$ jumps to the i-th command of the active procedure; the conditional goto $\mathsf{goto}\ [e]\ i,\ j$ jumps to the i-th command if e evaluates to true, and to the j-th otherwise. The procedure call $x := e(\overline{e})\ \mathsf{with}\ j$ is dynamic: the procedure identifier is obtained by evaluating the JSIL expression e. If the procedure terminates normally, control proceeds to the next command, and to the j-th command otherwise. External procedure calls can step outside the execution of a JSIL program, meaning that the procedures are implemented at the level of the analysis, where we have access to the JavaScript memory, instead of in JSIL. Such external calls are used to model the JavaScript `eval` function. It then becomes simpler to parse and evaluate the program given as string to the `eval` function at the analysis level. Procedure application receives the procedure identifier and a *JSIL list* containing the procedure parameters. The argument collection command returns the values of the arguments with which the current procedure was called.

A JSIL *procedure*, $proc \in \mathtt{Proc}$, is of the form $\mathsf{proc}\, f(\overline{x})\{\overline{c}\}$, where f is its identifier, $\overline{x}$ are its formal parameters, and its body $\overline{c}$ is a sequence of JSIL commands. Procedures can return either normally or in error mode, using the return and throw commands, respectively. A JSIL program $p \in \mathcal{P}$ is a set of top-level procedures, and its entry point is always the special procedure main.

```
proc evalUnop(op, v){
        goto [op = "-"] min r1;
min:    ret := -v;
        return;
r1:     goto [op = "not"] not r2;
not:    ret := not v;
        return;
r2:     goto [op = "abs"] abs r3;
abs:    ret := abs(v);
        return;
r3:     ret := "Error" ("UnOp");
        throw
};
```

```
proc evalBinop(op, v1, v2){
        goto [op = "+"] plu r1;
plu:    ret := v1 + v2;
        return;
r1:     goto [op = "-"] min r2;
min:    ret := v1 - v2;
        return;
r2:     goto [op = "or"] orl r3;
orl:    ret := v1 or v2;
        return;
r3:     goto [op = "and"] anl r4;
anl:    ret := v1 and v2;
        return;
r4:     ret := "Error" ("BinOp");
        throw
};
```

Figure 3.3.: Fragment of Expression Evaluator implemented in JSIL

JSIL as an IR for JavaScript Analysis. JSIL has all of the constructs needed to precisely capture and reason about the entire ES5 standard [27]. In particular: **(1)** the dynamic features of JavaScript (extensible objects, dynamic property access, and dynamic procedure calls[1]) are native in JSIL; **(2)** the intricate control flow patterns of JavaScript statements (e.g. `switch` and `try—catch`) can be expressed in the JSIL goto-based control flow; **(3)** the `eval` statement and the `Function` constructor, which require parsing of JavaScript code, are modelled via external procedure calls; **(4)** the `arguments` object is modelled with the help of the argument collection command; and **(5)** the internal properties of JavaScript objects are modelled using object metadata, streamlining the performance of JaVerT 2.0 on compiled JavaScript code.

Example. We illustrate the usage of the JSIL language in Figure 3.3, using the example of the expression evaluator. To this end, we give a stylised JSIL compilation of the `evalUnop` (left) and `evalBinop` (right) functions, associating each function with its corresponding JSIL procedure. In JSIL, the control flow is modelled using `gotos`; hence, the switch statements present in the original functions are mapped into a sequence of conditional `gotos`. For instance, given the command `goto [op = "-"] min r1`, the execution jumps to the label `min` if `op` is equal to `"-"`, and to the label `r1` otherwise. We use the JSIL arithmetic and logical operators to implement operations supported by the `evalUnop` and `evalBinop`. Note that a JSIL procedure must either finish executing successfully (via the `return` command) or with an error (via the `throw` command). In both cases, the value that is returned is the value of the dedicated variable `ret`. Note that, in the error case, the returned value is the error object created with the JSIL `Error` constructor (via "`Error`"(...)).

3.2.2. JSIL Semantics

Instead of defining two separate semantics for JSIL, a concrete one for concrete execution and a symbolic one for symbolic execution, we define a single *general semantics* that can be instantiated with either a concrete or a symbolic state signature. This general semantics is the bedrock for both the formal development and the implementation of JaVerT 2.0, avoiding redundancy in both the formalism and the implementation.

The general JSIL semantics describes the behaviour of JSIL commands in terms of a general *state*

[1]In JavaScript, one can call a function using `obj[f]()`, where `f` is resolved dynamically.

$$\frac{\text{cmd}(i) = x := [e_1, e_2] \quad \mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (-,-,\mathtt{v}) \quad \Sigma'' = \mathcal{SS}(\Sigma', x, \mathtt{v})}{\langle \Sigma, \mathtt{cs}, i \rangle \rightsquigarrow \langle \Sigma'', \mathtt{cs}, i{+}1 \rangle} \ \textsc{Property Access}$$

$$\frac{\text{cmd}(i) = \text{assume}\,(e) \quad \Sigma' = \mathcal{A}sm(\Sigma, e)}{\langle \Sigma, \mathtt{cs}, i \rangle \rightsquigarrow \langle \Sigma', \mathtt{cs}, i{+}1 \rangle} \ \textsc{Assume}$$

$$\frac{\text{cmd}(i) = \text{goto}\ [e]\ j,\ k \quad \Sigma' = \mathcal{A}sm(\Sigma, e)}{\langle \Sigma, \mathtt{cs}, i \rangle \rightsquigarrow \langle \Sigma', \mathtt{cs}, j \rangle} \ \textsc{Cond. Goto - True}$$

Figure 3.4.: General semantics of commands, non-failing transitions (excerpt): $\langle \Sigma, \mathtt{cs}, i \rangle \rightsquigarrow \langle \Sigma', \mathtt{cs}', j \rangle$

signature: that is, a set of state functions, reminiscent of *local actions* in SL [24, 18], which capture the fundamental ways in which JSIL programs interact with JSIL states; for example, evaluating an expression or retrieving the value of an object property. This general state signature can then be instantiated to obtain a specific JSIL semantics. In JaVerT 2.0, we provide two state instantiations: *concrete* and *symbolic*, respectively obtaining the concrete and symbolic semantics of JSIL.

We require general states to contain a variable store $\mathcal{S}to : \mathcal{X} \rightharpoonup \mathbb{V}$, mapping program variables $x \in \mathcal{X}$ to general values. Stores have two functions associated with them: a **store getter** (GetStore), $\mathcal{GS}(\Sigma)$, which returns the store associated with the state Σ; and a **store setter** (SetStore), $\mathcal{SS}(\Sigma, x, \mathtt{v})$, which returns the state obtained from Σ by updating the value of x to $\mathtt{v}$ in the store of Σ.

In Figure 3.4, we show a simplified version of three rules of the general JSIL semantics: Property Access, Assume and Cond Goto - True. The rules of the JSIL semantics have the form $\langle \Sigma, \mathtt{cs}, i \rangle \rightsquigarrow \langle \Sigma', \mathtt{cs}', j \rangle$, where: **(1)** Σ and Σ' denote the current and the next JSIL states; **(2)** $\mathtt{cs}$ and $\mathtt{cs}'$ denote the current and the next JSIL call stacks[2]; and **(3)** i and j denote the indexes of the current and the next commands to be executed.

Below, we describe the rules given in Figure 3.4:

[Property Access] If the current JSIL command is a property access, $x := [e_1, e_2]$, the general semantics uses the GetCell state function, $\mathcal{GC}$, to obtain the value, $\mathtt{v}$, associated with the property denoted by e_2 in the object at the location denoted by e_1. Then, the semantics uses the SetStore state function, $\mathcal{SS}$, to set the variable x in the current store to the value $\mathtt{v}$, obtaining the new state Σ''. Finally, the control is transferred to the next command at index $i + 1$.

[Assume] If the current JSIL command is an assume, assume (e), the general semantics uses the Assume state function, $\mathcal{A}sm$, to extend the current state with the assumption being provided, obtaining the new state Σ'. Note that this state function is not total, meaning that it produces no result when the expression being assumed is inconsistent with the current state Σ. Analogously to the previous rule, the next index is simply set to $i + 1$.

[Cond Goto - True] If the current JSIL command is a conditional goto, goto $[e]\ j,\ k$, the general semantics performs two transitions, one for the **true** case and one for the **false** case. In **true** case, the semantics uses the Assume state function, $\mathcal{A}sm$, to extend the current state with the information that the conditional guard of the goto, e, is true. Accordingly, the index of the next command to be executed is set to j, corresponding to the then-case of the conditional goto.

[2] Intuitively, call stacks are used to keep track of the stores of the functions whose execution has not yet terminated, so that those stores can be reinstated once the control is transferred back to their respective functions.

$$\dfrac{\sigma = (s,h) \quad l = \mathcal{E}v_c(s,e_1) \quad p = \mathcal{E}v_c(s,e_2) \quad h = - \uplus (l,p) \mapsto v \quad r = (l,p,v)}{\mathcal{GC}(\sigma, e_1, e_2) \rightsquigarrow_{\mathsf{c}} \sigma, r}\ \textsc{GetCell - Found}$$

$$\dfrac{\sigma = (s,h) \quad s' = s[x \mapsto v] \quad \sigma' = (s', h)}{\mathcal{SS}_{\mathsf{c}}(\sigma, x, v) \triangleq \sigma'}\ \textsc{SetStore}$$

$$\dfrac{\mathcal{E}v_{\mathsf{c}}(\mathcal{GS}(\sigma), e) = \mathsf{true}}{\mathcal{A}sm_{\mathsf{c}}(\sigma, e) \triangleq \sigma}\ \textsc{Assumption}$$

Figure 3.5.: Concrete JSIL State Rules

Note that our general semantics is non-deterministic, as it allows multiple transitions to take place. For instance, in the Property Access rule, the GetCell state function can produce several outcomes, especially in the presence of symbolic states in which there is an excessive amount of branching. In the following, we detail the concrete and symbolic instantiations of our general JSIL semantics.

JSIL Concrete Semantics. The concrete semantics allows us to run JSIL programs concretely. This is essential for ensuring that the general semantics captures the intended behaviour of the language. Furthermore, it allows us to test our infrastructure against the ECMAScript official test suite, Test262 [26], by first compiling it to JSIL and then executing it concretely. In this way, we establish trust in the compilation.

The concrete JSIL state signature defines both the structure of concrete JSIL states as well as the functions required to interact with those states. JSIL concrete states are of the form (s, h), consisting of a variable store, $s \in \mathcal{S}to : \mathcal{X} \rightharpoonup \mathcal{V}$, mapping program variables to their respective concrete values, which are JSIL literals, and an object heap, $h \in \mathcal{H} : (\mathcal{L} \times \mathcal{V}) \rightharpoonup \mathcal{V}$, mapping pairs of object locations and property names to their corresponding values. Figure 3.5 gives the concrete state rules for GetCell, SetStore, and Assumption, used by the general semantic rules shown in Figure 3.4. We describe each of these rules below.

[GetCell - Found] Using the concrete expression evaluation function, $\mathcal{E}v_c$, the concrete GetCell function, $\mathcal{GC}(\sigma, e_1, e_2)$, evaluates the expressions e_1 and e_2, obtaining the location l and property p. Then, it looks up the the value of the property p in the object pointed to by l in the heap h, obtaining the value v. We use the $\uplus$ operator to split the heap into two disjoint parts. Finally, the semantics returns a pair consisting of the unchanged concrete state, σ, and a triple, (l, p, v), with the computed location, property, and value. We omit the GetCell - Not Found transition for brevity.

[SetStore] The concrete SetStore function, $\mathcal{SS}_{\mathsf{c}}(\sigma, x, v)$, simply updates the store of the current state σ by setting the value of the variable x to v, thus obtaining a new state σ'.

[Assumption] The concrete Assume function, $\mathcal{A}sm_{\mathsf{c}}(\sigma, e)$, simply checks whether its argument e evaluates to true. If it does, it returns the current state σ unchanged. If it does not, no transition is provided.

JSIL Symbolic Semantics. The symbolic semantics enables the symbolic execution of JSIL programs. We use the symbolic execution engine of JaVerT 2.0, for instance, for symbolic testing purposes. Symbolic testing allows us to establish the bounded correctness of important functional properties and

GetCell - Found	SetStore	Assumption
$$\dfrac{\hat{\sigma} = (\hat{s}, \hat{h}, \pi) \quad \hat{l}, \hat{p} = \mathcal{E}v_{\mathsf{s}}(\hat{s}, e_1), \mathcal{E}v_{\mathsf{s}}(\hat{s}, e_2) \quad \pi \vdash \hat{p} = \hat{p}' \quad \hat{h}(\hat{l}, \hat{p}') = \hat{v}}{\mathcal{GC}(\hat{\sigma}, e_1, e_2) \rightsquigarrow_s (\hat{s}, \hat{h}, \pi), (\hat{l}, \hat{p}', \hat{v})}$$	$$\dfrac{\hat{\sigma} = (\hat{s}, \hat{h}, \pi) \quad \hat{s}' = \hat{s}[\hat{x} \mapsto \hat{v}] \quad \hat{\sigma}' = (\hat{s}', \hat{h}, \pi)}{\mathcal{SS}_{\mathsf{s}}(\hat{\sigma}, \hat{x}, \hat{v}) \triangleq \hat{\sigma}'}$$	$$\dfrac{\hat{\sigma} = (\hat{s}, \hat{h}, \pi) \quad \hat{b} = \mathcal{E}v_{\mathsf{s}}(\hat{s}, e) \quad \mathcal{S}at_{\mathsf{s}}(\hat{\sigma}, \pi \wedge \hat{b})}{\mathcal{A}sm_{\mathsf{s}}(\hat{\sigma}, e) = (\hat{s}, \hat{h}, \pi \wedge \hat{b})}$$

Figure 3.6.: Symbolic JSIL State Rules

find bugs. A symbolic test contains symbolic inputs instead of concrete ones, and include assertions to describe functional properties that must be satisfied by the program output.

We instantiate general values to *symbolic values*, $\hat{v} \in \hat{\mathcal{V}} \triangleq v \mid \hat{x} \mid \ominus\ \hat{v} \mid \hat{v} \oplus \hat{v}$, and write $\hat{n}$, $\hat{b}$, $\widehat{str}$, $\hat{l}$, and $\hat{p}$, to denote, respectively, symbolic numbers, booleans, strings, locations, and property names. A symbolic JSIL state, $\hat{\sigma} = (\hat{s}, \hat{h}, \pi)$, consists of a symbolic store $\hat{s}$, symbolic heap $\hat{h}$, and a path condition π. Symbolic stores are obtained from their concrete counterparts by allowing symbolic values in place of concrete ones. Symbolic heaps, $\hat{h} \in \hat{\mathcal{H}} : ((\mathcal{L} \uplus \hat{\mathcal{L}}) \times \hat{\mathcal{V}}) \rightharpoonup \hat{\mathcal{V}}$, map pairs of object locations (both symbolic and concrete) and symbolic values to symbolic values. Path conditions [10] bookkeep the constraints on the symbolic variables that led the execution to the current symbolic state.

Figure 3.6 gives the concrete state rules for GetCell, SetStore, and Assumption, used by the general semantic rules shown in Figure 3.4. We describe each of these rules in the following.

[GetCell - Found] Using the symbolic expression evaluation function, $\mathcal{E}v_s$, the symbolic GetCell function, $\mathcal{GC}(\hat{\sigma}, e_1, e_2)$, evaluates the expressions e_1 and e_2, obtaining the symbolic location $\hat{l}$ and symbolic property $\hat{p}$. Then, it tries to find the property $\hat{p}$ in the object pointed by $\hat{l}$ in the heap $\hat{h}$. The property is found if we can prove that it is equal to one of the existing properties $\hat{p}'$, which has value $\hat{v}$. In that case, the function returns a pair containing the unchanged symbolic state, $\hat{\sigma}$, and a triple, $(\hat{l}, \hat{p}', \hat{v})$, containing the computed location, property and value. We omit the GetCell - Not Found rule for brevity.

[SetStore] The symbolic SetStore function, $\mathcal{SS}_{\mathsf{s}}(\hat{\sigma}, \hat{x}, \hat{v})$, is analogous to its concrete counterpart. It simply updates the store of the current symbolic state $\hat{\sigma}$ by setting the value of the variable $\hat{x}$ to $\hat{v}$, obtaining a new state $\hat{\sigma}'$.

[Assumption] The symbolic Assume function, $\mathcal{A}sm_{\mathsf{s}}(\hat{\sigma}, e)$, first evaluates the expression e, obtaining a symbolic boolean $\hat{b}$. Then, the function conjuncts $\hat{b}$ with the current path condition π, and, if the obtained formula is satisfiable, sets it as the path condition of the resulting symbolic state. Otherwise, no transition is provided. Finally, the function returns the new state with the store and heap unchanged and the updated path condition.

The symbolic semantics of JaVerT 2.0 comes with two correctness guarantees:

- *Directed Soundness:* establishes that each symbolic execution trace over-approximates all concrete traces that follow its execution path and whose initial concrete states are over-approximated by the initial symbolic state;

- *Directed Completeness:* establishes that each symbolic execution trace has at least one concretisation.

While directed soundness is essential for bounded-verification, directed completeness is required for guaranteeing the absence of false positive bug reports. The formal results are given in [33, 35]

4. Event Semantics

We introduce the Event Semantics, a minimal formalism able to capture the essence of three fundamental, complex APIs—DOM Events [139], JS Promises [28], and JS async/await [29]. The DOM API first became available in 1998 and defines an interface for dynamically updating the content of a webpage through the use of a scripting language such as JavaScript. The API evolved over the years, eventually introducing DOM Events to model event-related features such as event handler registration/deregistration and event dispatch. In the meantime, the JavaScript language, which is regulated by the ECMAScript standard, became the *de facto* language of the Web. In order to provide better asynchronous programming features and avoid the so-called *callback hell* [37], the ECMAScript standard introduced JS Promises in its 6th version. Later, in the 8th version of the ECMAScript standard, the `async` and `await` operators were introduced to the language to enable a higher level abstraction over JS Promises.

Most Web applications rely on these three event-driven APIs and, due to their high level of complexity, developers introduce bugs caused by the misuse of such APIs [94]. Previous works [100, 83] design formal models targeting a specific API. Our challenge is to have a unified Event Semantics (onward: E-semantics) that is rather general and formalises all event-related features from the DOM Events, JS Promises and JS async/await APIs.

Outline. We start by giving a motivating example (§4.1) and then explain the parametric construction of the E-semantics (§4.2). Next, we introduce our event syntax (§4.3). We then illustrate how JaVerT.Click performs symbolic analysis of event-based JavaScript programs using the motivating example (§4.4). We build JaVerT.Click on top of JaVerT 2.0 [35] (cf. Chapter 3), a symbolic analysis tool for JavaScript which supports whole-program symbolic testing, verification and bi-abduction. JaVerT.Click adds an E-semantics module to JaVerT 2.0 that can be instantiated either with a concrete or a symbolic underlying language semantics. However, for clarity purposes, we choose to present the concrete (§4.5) and the symbolic (§4.6) E-semantics separately. We conclude by providing a correctness result for the symbolic E-semantics with respect to the concrete E-semantics.

4.1. Motivating Example

We use a simple example to illustrate the complexity of event-based JavaScript programs. Consider the client of the DOM API shown in Figure 4.1, including the HTML file (left) and its associated JavaScript code (right). The HTML file illustrates a simple webpage containing a single element to which we can associate events. The JavaScript code uses DOM functions to dynamically update the content of the webpage and trigger events. We write a symbolic test as shown in Figure 4.2 for the program given in Figure 4.1. In order to distinguish JavaScript and DOM features, we highlight DOM functions in purple.

Initialisation code. The HTML file contains, in its body, a `div`[1] element with an `id` attribute set to "`name`". The JavaScript code, which we assume to run after the HTML file is loaded, enables two possible types of events on the `div` element: "`init`" and "`print`". The "`init`" event aims at assigning a name to the `person` object and the "`print`" event aims at printing the assigned name to the webpage.

More concretely, in lines 1-2, the JS script declares the variables `person` and `elem`, and assigns to `elem` the DOM `div` element which has `id` equal to "`name`". This is possible by using the function `getElementById` provided by the DOM `Document` interface. Next, in lines 4-10, the script declares two functions: `hdlr1`, which initialises the variable `person`, and `hdlr2`, which changes the inner HTML of `elem` to display the value of `person.name`. Consequently, the value of `person.name` should appear inside the `div` element in the webpage. Finally, in lines 12-13, we add `hdlr1` and `hdlr2` as handlers for the "`init`" and "`print`" events by using the DOM function `addEventListener`, which is exposed by the DOM `EventTarget` interface.

```html
<!DOCTYPE html>
<html>
<body>

<h1>Simple HTML Document</h1>

<div id="name"></div>

</body>
</html>
```

```javascript
1  var person;
2  var elem = document.getElementById("name");
3
4  function hdlr1() {
5    person = {name:"Mary"};
6  }
7
8  function hdlr2() {
9    elem.innerHTML = person.name;
10 }
11
12 elem.addEventListener("init", hdlr1);
13 elem.addEventListener("print", hdlr2);
```

Figure 4.1.: Event-based client of DOM API, including HTML (left) and JavaScript code (right)

Symbolic test. In Figure 4.2, we show a simple symbolic test for the given initialisation code introduced in Figure 4.1. The goal of the test is to create two symbolic DOM events and dispatch both events on the element `elem`. We make use of the `symbStr` symbolic execution primitive provided by JaVerT.Click to create symbolic strings. First, in lines 1-2, we create two symbolic strings `et1` and `et2` representing event types. Next, in lines 4-5, we create two DOM events by invoking the `Event` constructor passing the event types `et1` and `et2` as arguments. Finally, in lines 7-9, we first obtain the DOM element `elem` by using its id "`name`", and dispatch the two events `e1` and `e2` on the DOM element `elem` by invoking the `dispatchEvent` function exposed by the `EventTarget` DOM interface. In summary, during the dispatch, the related handlers (here, any handlers for `e1` and `e2`) are executed one by one. Because the events `e1` and `e2` are symbolic, there are multiple possible outcomes.

By running the test in JaVerT.Click, we are able to discover a bug in the initialisation code given in Figure 4.1. Whenever the handler `hdlr2` is executed before handler `hdlr1`, the execution leads to a native JavaScript type error, as a result of trying to access the property `name` of the object `person` when `person` is not yet initialised.

JaVerT.Click gives a failing model as output, which contains, for each symbolic variable, sets of concrete values that cause the test to fail. For this particular test, JaVerT.Click gives the following

[1] The HTML `div` tag is useful to create divisions in the webpage.

```
var et1 = symbStr();
var et2 = symbStr();

var e1 = new Event(et1);
var e2 = new Event(et2);

var elem = document.getElementById("name");
elem.dispatchEvent(e1);
elem.dispatchEvent(e2);
```

Figure 4.2.: Symbolic test for DOM client given in Figure 4.1.

failing model:

$$\{(\texttt{et1} = \text{“}\texttt{print}\text{”}, \texttt{et2} = \#\texttt{et2}), (\#\texttt{et1} \notin \{\text{“}\texttt{init}\text{”}, \text{“}\texttt{print}\text{”}\}, \texttt{et2} = \text{“}\texttt{print}\text{”})\}.$$

This means that, for these two cases, the test fails due to an unhandled JavaScript type error. We use #`et2` to denote any concrete string value. The dispatch of an event of type “`print`” without a previous dispatch of an event of type “`init`” causes the issue, as handler `hdlr2` is triggered before handler `hdlr1`. The use of a symbolic execution engine such as JaVerT.Click helps to find such bug that can go unnoticed during the development of real-world Web applications. While there are other symbolic execution tools focused on event-driven applications [84, 123], we are not aware of any that supports multiple event-based APIs and allows to represent events symbolically. JaVerT.Click is the first to provide built-in support for multiple event-based APIs and to allow for the use of symbolic events.

4.2. Parametric Construction

We define the E-semantics parametrically, as a layer on top of the semantics of a given underlying language (L), thus focussing only on event-related details and filtering out any clutter potentially introduced by the L-semantics. In JaVerT.Click, we instantiate the E-semantics with JSIL (JaVerT intermediate language for JavaScript). The E-semantics interacts with the L-semantics by exposing a set of *event primitives*, which correspond to the fundamental operations underpinning the targeted APIs, such as event handler registration and asynchronous event dispatch. Our parametric construction also makes the E-semantics easily extensible with support for further event-based APIs. In Figure 4.3, we show how our E-semantics is parametric on the underlying language both in terms of configurations (left) and transitions (right).

Parametric Configurations. Every configuration of the E-semantics (onward: E-configuration) contains a configuration of the underlying language. This means that an E-configuration is parametric on an L-configuration. Note that the language configuration can be either concrete (lc) or symbolic ($\widehat{lc}$). Hence, the E-semantics is generic in the sense that it can be instantiated with either a concrete or a symbolic underlying language semantics, respectively leading to a concrete and symbolic E-semantics.

Parametric Transitions. A transition of the E-semantics has the form $\epsilon c \rightsquigarrow_{\mathsf{E}} \epsilon c'$, where ϵc and $\epsilon c'$ are E-configurations. Transitions of the E-semantics are built on top of transitions of the underlying language semantics, and they can be either concrete or symbolic. Hence, if the L-semantics makes a step, the E-semantics follows accordingly. We use $lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'$ to denote a transition of the language semantics. Depending on the primitive p returned by the L-semantics, the E-semantics knows whether or not the currently processing command is event-related. If so, the E-semantics takes action accordingly. Otherwise, the E-semantics simply updates its L-configuration lc to lc'.

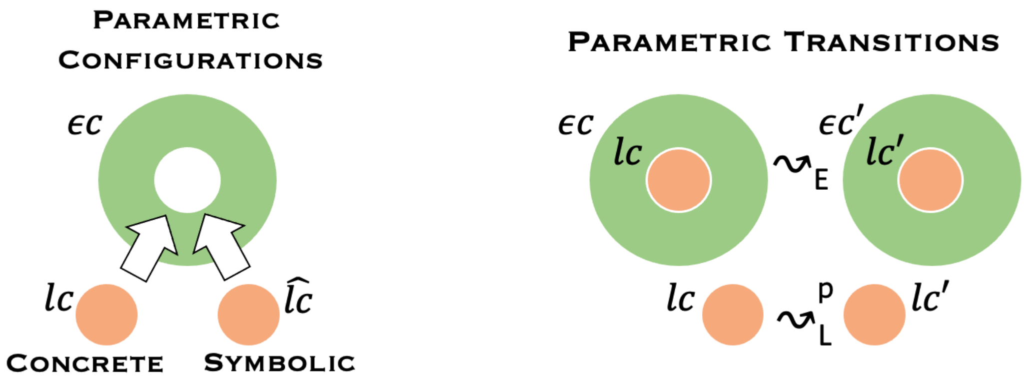

Figure 4.3.: The parametric construction of the E-semantics

4.3. Event Syntax

The event syntax is given in Figure 4.4. We highlight in blue the elements that are provided by the E-semantics. The E-semantics inherits its *values*, $v \in \mathcal{V}$, from the corresponding L-semantics: for example, if the L-semantics is concrete, these values will be concrete; analogously, if it is symbolic, they will be symbolic. In the meta-theory, we assume that the L-values contain: a distinguished set of unique *event types*, $e \in \mathcal{E}$, intuitively corresponding to, for example, "`init`" or "`print`" in the DOM; and a distinguished set of unique *function identifiers*, $f \in \mathcal{F}$. In the implementation, we represent both as strings. For simplicity, we onward refer to event types as *events*. Our modelling of events is guided by the DOM, in the sense that each event is associated with a list of *handlers*: that is, the functions that should be executed when that event is triggered; this information is kept by the E-semantics in *handler registers*, $h \in \mathcal{H}$.

The E-semantics, expectedly, needs to be aware of the configurations of the underlying language (L-configurations), $lc \in \mathcal{LC}$, but sees them as a black box and interacts with them only through an interface, presented shortly. To model correctly the synchronous dispatch of the DOM and the asynchronous wait of the JS `await`, we also require boolean predicates on L-configurations, $\rho \in \mathcal{P}$.

The L-semantics communicates with the E-semantics via event primitives, $\mathrm{p} \in \mathrm{P}$. In particular, $\cdot$ is used to indicate that the current command is not event-related; **addHdlr** and **remHdlr**, respectively, allow us to add and remove handlers for a given event, whereas **sDispatch** and **aDispatch**, respectively, allow us to dispatch events either *synchronously* (corresponding to the DOM programmatic dispatch) or *asynchronously* (corresponding to a user event, such as clicking a button on a Web page). These four primitives are used in the modelling of DOM Events (cf. §6.2). Additionally, we support asynchronous computation scheduling via the schedule primitive, required for JS Promises (cf. §6.3), and an asynchronous wait via the await primitive, required for JS `await` (cf. §6.4). Note that by defining a set of only 6 primitives we are able to model event-based features from three different APIs:

Values	**Event Types**	**Function Ids**	**L-Confs**	**Conf. Preds**
$v \in \mathcal{V}$	$e \in \mathcal{E} \subset \mathcal{V}$	$f \in \mathcal{F} \subset \mathcal{V}$	$lc \in \mathcal{LC}$	$\rho \in \mathcal{P} : \mathcal{LC} \to \mathbb{B}$

Event Primitives

$\mathrm{p} \in \mathrm{P} := \cdot \mid \mathsf{addHdlr}\langle e, f\rangle \mid \mathsf{remHdlr}\langle e, f\rangle \mid \mathsf{sDispatch}\langle e, vs\rangle \mid \mathsf{aDispatch}\langle e, vs\rangle \mid \mathsf{schedule}\langle f, vs\rangle \mid \mathsf{await}\langle v, \rho\rangle$

Handler Registers	**Continuations**	**Continuation Queues**	**E-Configurations**
$h \in \mathcal{H} : \mathcal{E} \rightharpoonup \overline{\mathcal{F}}$	$\kappa \in \mathcal{K} := (f, vs) \mid (c, \rho)$	$q \in \mathcal{Q} : \overline{\mathcal{K}}$	$\epsilon c \in \mathcal{EC} : \mathcal{LC} \times \mathcal{H} \times \mathcal{Q}$

Figure 4.4.: Events Syntax

DOM Events, JS Promises and JS async/await.

All three targeted APIs work with an underlying queue of computations: for the DOM, this queue is implicitly formed by event dispatch; for JavaScript promises and `async/await`, this queue is the job queue of JavaScript. We model these queues as a unified *continuation queue*, $q \in \mathcal{Q}$, which is, essentially, a list of *continuations*, $\kappa \in \mathcal{K}$, which describe how the execution of the E-semantics is to proceed. We consider two types of continuations: handler-continuations and yield-continuations. A *handler-continuation* is a pair, (f, vs), essentially stating that the handler f is be to be executed with arguments vs. When an event is dispatched via **sDispatch** or **aDispatch**, the respective handler-continuations are put in the handler queue. A *yield-continuation* is a pair, (lc, ρ), stating that the L-configuration c has been suspended and can be re-activated once the predicate ρ holds.

Finally, the E-semantics configurations, (E-configurations), $\epsilon c \in \mathcal{EC}$, consist of: an L-configuration; a handler register; and a continuation queue.

Using the E-semantics in JavaScript. Our JS reference implementations of the event-related APIs interact with the E-semantics via *JS wrapper functions*, one per event primitive; we denote, for example, the wrapper function of the addHdlr primitive by `__addHdlr`, and the others analogously. Calls to these wrapper functions are intercepted by the underlying JavaScript implementation, which is then required to construct the corresponding primitive and pass it on to the E-semantics. In JaVerT.Click, these wrapper functions resolve to JSIL functions with dedicated identifiers, whose calls are then intercepted appropriately by the JSIL semantics.

4.4. Symbolic Analysis of Motivating Example

In Figure 4.5, we show a subset of the E-configurations computed during the symbolic execution of our motivating example introduced in §4.1. We distinguish initialisation code (see Figure 4.1) from symbolic testing code (see Figure 4.2). The initial E-configuration contains: an L-configuration lc, an empty handler register h and an empty continuation queue q. We omit the details of the L-configuration and focus our discussion on the changes to the handler register and continuation queue.

We start by analysing the initialisation code given in Figure 4.1. Internally, our implementation of the DOM function `addEventListener` calls the `__addHdlr` wrapper function, which in turn generates the `addHdlr` event primitive of the E-semantics. The generation of the `addHdlr` primitive during the addition of the handlers `hdlr1` and `hdlr2` via `addEventListener` (step `1` in Figure 4.5) causes the handler register h to be updated with two new entries: event "`init`" is mapped to a list of handlers containing only `hdlr1` and the event "`print`" is mapped to a list of handlers containing only `hdlr2`.

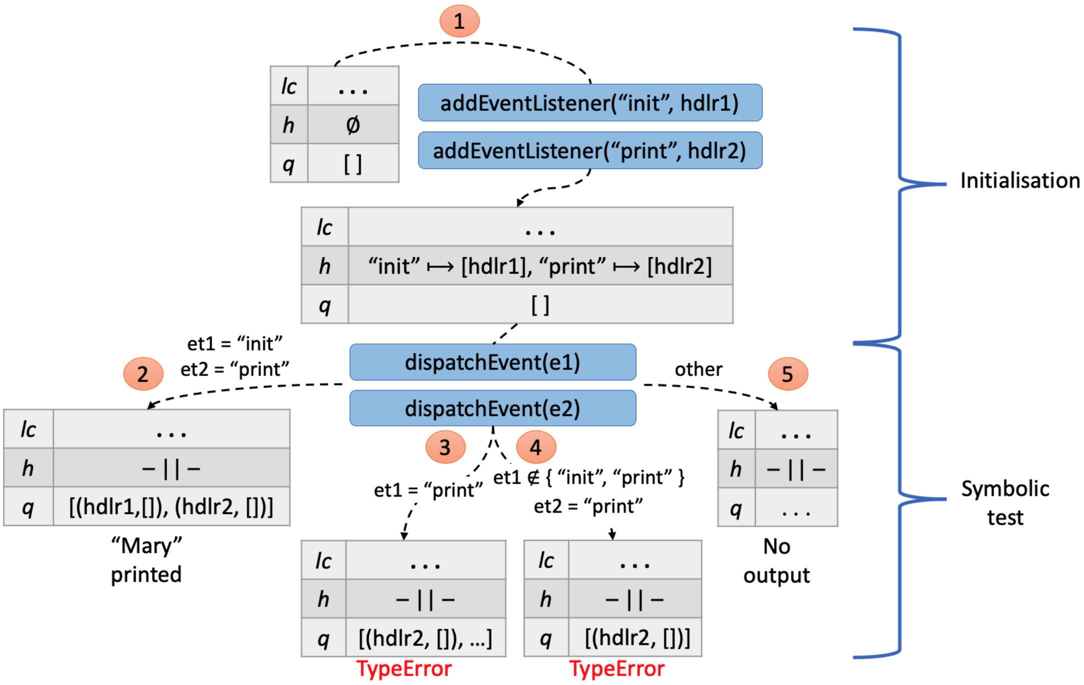

Figure 4.5.: E-configurations for motivating example.

Finally, we run the symbolic test, which dispatches the two events `e1` and `e2` using the DOM function `dispatchEvent`. Internally, our implementation of the `dispatchEvent` function calls the `__sDispatch` wrapper function, which generates the `sDispatch` event primitive. The reason for using `sDispatch` instead of `aDispatch` is that the DOM programatic dispatch (enabled via `dispatchEvent`) is always synchronous. No changes are made to the handler register during an event dispatch. The continuation queue, however, is updated with the respective handlers associated to the events `e1` and `e2`. At this stage, the execution of JaVerT.Click branches and there are four possible cases which are listed below:

Step 2**:** `et1` = "`init`" **and** `et2` = "`print`"**.** The handler `hdlr1` is executed before `hdlr2`, meaning that the execution terminates successfully and the string "`Mary`" becomes visible on the webpage.

Step 3**:** `et1` = "`print`"**.** The handler `hdlr2` is executed first leading to a native JavaScript type error, as `hdlr2` attempts to read the `name` property of `person`, which is not yet initialised.

Step 4**:** `et1` ∉ {"`init`", "`print`" } **and** `et2` = "`print`"**.** Same output as above; the variable `person` has not been initialised during the execution of the handler `hdlr2`, leading to a type error.

Step 5**: other.** Handlers `hdlr1` and `hdlr2` are not triggered, thus the execution terminates successfully but with no visible changes on the webpage.

4.5. Concrete E-semantics

A concrete E-semantics is built on top of a concrete L-semantics and requires the L-semantics to provide an interface in order to update L-configurations. Besides calling the functions provided by the L-semantics interface, the concrete rules of the E-semantics also call auxiliary functions to handle event-related operations, such as event handler registration and event dispatch. In the following, we introduce first the L-semantics interface (§4.5.1) and then the E-semantics auxiliary functions (§4.5.2). Finally, we present the rules of the concrete E-semantics (§4.5.3).

4.5.1. Concrete L-semantics Interface

Each configuration of the E-semantics depends on a configuration of the L-semantics, which is required to have: a *store component*, describing the variable store of L; and a *memory component*, describing the memory on which L-programs operates; and a *control flow component*, describing how the L-execution is to proceed. For example, a concrete JSIL configuration, $\langle s, h, cs, i \rangle$, consists of: a variable store s (the store component); a heap h (the memory component); and a call stack cs for capturing nested function calls and the index of the next command to be executed, i (the control flow component). The L-semantics also needs to provide an interface the following functions: initialConf, mergeConfs, final, suspend and splitReturn.

1. $\mathsf{initialConf}(lc, (f, vs)) = lc'$: creates a new configuration lc' which has the memory of the given configuration lc and the control flow/store components required for starting the execution of the handler f with arguments vs. The E-semantics uses this function to set up a new configuration to execute an event handler.

2. $\mathsf{mergeConfs}(lc, lc') = lc''$: merges two configurations lc and lc' into a new configuration lc'', which has the memory component of lc and the control flow/store components of lc'. In particular, in JSIL, given $lc = \langle s, h, cs, i \rangle$ and $lc' = \langle s', h', cs', i' \rangle$, the call to $\mathsf{mergeConfs}(lc, lc')$ would result in $\langle s', h, cs', i' \rangle$. The E-semantics uses this function to merge the configuration resulting from the synchronous execution of event handlers (lc) with the caller configuration (lc').

3. $\mathsf{final}(lc) =$ `true`, if lc cannot be executed further; and $=$ `false` otherwise. The E-semantics uses this function to determine if the current computation is complete in order to process the next continuation in the continuation queue.

4. $\mathsf{suspend}(lc) = lc'$: suspends the given configuration lc by marking it as final, generating the configuration lc'. The E-semantics uses this function to suspend the current language configuration before handling a synchronous event dispatch.

5. $\mathsf{splitReturn}(lc, \mathsf{v}) = (lc_r, lc_a)$: splits the current configuration lc into lc_r and lc_a. The configuration lc_r is obtained by pausing the execution of the current procedure, and the configuration lc_a contains the remainder of the execution of the current procedure. Consider the following JavaScript code snippet. Function `g` (left) calls the asynchronous function `f` (right), which calls `await` on an arbitrary value `v`. Because of the `await` expression, the execution of function `f` needs to pause and function `g` must continue to execute. To process the call to `await`, the E-semantics uses the splitReturn function, which computes: **(1)** the configuration lc_r obtained from lc by setting up the control flow component as if the function `f` had returned the value `v`, so that function `g` can continue to execute normally, and **(2)** the configuration lc_a is obtained from lc by setting up the control flow component to only contain the remainder of the execution of `f`.

```
function g(){
  ...
  f()
  ...
}
```

```
async function f(){
  ...
  await v;
  ... // remainder of f
}
```

From now on, we use the prefix L. to denote a call to a function provided by the L-semantics interface.

4.5.2. Auxiliary Functions of the Concrete E-semantics

To improve the modularity and readability of our E-semantics, we implement four event-related operations in auxiliary functions, which are the following:

Add handler: $\mathcal{AH}(h, e, f)$ extends the handler register h with the handler f for an event e;

Remove handler: $\mathcal{RH}(h, e, f)$ removes the handler f for e from h;

Find handlers: $\mathcal{FH}(h, e)$ obtains the handlers associated with e in h; and

Continue with: $\mathcal{CW}_{\text{L}}(lc, \kappa)$ updates the L-configuration lc so that the continuation κ can be executed.

We give the formal definitions of these auxiliary functions in Figure 4.6. We write $+\!\!+$ to denote list concatenation; $h_o(e)$ to denote $h(e)$ if it is defined, and the empty list otherwise; and $l \setminus f$ to denote the list obtained from the list l by removing all occurrences of f.

Add Handler

$$\mathcal{AH}(h, e, f) \triangleq h\,[e \mapsto h_o(e) +\!\!+ [f]]$$

Find Handler

$$\mathcal{FH}(h, e) \triangleq h_o(e)$$

CW-Handler-Cont.

$$\mathcal{CW}_{\text{L}}(lc, (f, vs)) \triangleq \mathsf{L.initialConf}(lc, (f, vs))$$

Remove Handler

$$\mathcal{RH}(h, e, f) \triangleq \begin{cases} h\,[e \mapsto h(e) \setminus f], & \text{if } e \in \mathtt{dom}(h) \\ h, & \text{otherwise} \end{cases}$$

CW-Yield-Cont.

$$\frac{\rho(lc) = \mathsf{True}}{\mathcal{CW}_{\text{L}}(lc, (lc', \rho)) \triangleq \mathsf{L.mergeConfs}(lc, lc')}$$

Figure 4.6.: Concrete E-semantics: Auxiliary Functions

These definitions are straightforward except for $\mathcal{CW}_{\text{L}}$, which contains two possible cases; either the continuation is a handler-continuation or a yield-continuation. When given a handler-continuation, $\kappa = (f, vs)$, the $\mathcal{CW}_{\text{L}}$ function sets up the execution of the handler f with arguments vs by using the $\mathsf{initialConf}$ function of the L-semantics interface. When given a yield-continuation, $\kappa = (lc', \rho)$, the $\mathcal{CW}_{\text{L}}$ function requires the predicate ρ to hold for the current L-configuration lc, in which case it merges the two configurations using the $\mathsf{mergeConfs}(lc, lc')$ function of the L-semantics interface.

4.5.3. Concrete E-semantics Rules

We now give the concrete E-semantics transitions in Figure 4.7, which are of the form $\epsilon c \leadsto_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$, where ϵc and $\epsilon c'$, respectively, are the configurations before and after the computed step, and $\epsilon\alpha$ is an environment action. Environment actions are used to model events triggered by the environment, such as user UI-events and network events. They have the grammar $\epsilon\alpha ::= \cdot \mid \mathsf{fire}\langle e, vs\rangle$, where $\cdot$

represents no environment action and $\mathsf{fire}\langle e, vs\rangle$ represents the triggering of the event e with values vs. For clarity, we elide $\cdot$ in the transitions.

Language Transition
$$\frac{lc \rightsquigarrow^{\cdot}_{\mathrm{L}} lc'}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', h, q\rangle}$$

Add Handler
$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{addHdlr}\langle e, f\rangle}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', \mathcal{AH}(h, e, f), q\rangle}$$

Remove Handler
$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{remHdlr}\langle e, f\rangle}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', \mathcal{RH}(h, e, f), q\rangle}$$

Synchronous Dispatch
$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{sDispatch}\langle e, vs\rangle \quad [f_i \mid_0^n] = \mathcal{FH}(h, e) \qquad q' = [(f_i, vs) \mid_{i=0}^n] \quad lc'' = \mathrm{L}.\mathsf{suspend}(lc')}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc'', h, q' +\!\!+ [(lc', (\lambda lc.\mathsf{True}))] +\!\!+ q\rangle}$$

Asynchronous Dispatch
$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{aDispatch}\langle e, vs\rangle \qquad [f_i \mid_0^n] = \mathcal{FH}(h, e) \quad q' = [(f_i, vs) \mid_{i=0}^n]}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', h, q +\!\!+ q'\rangle}$$

Schedule
$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{schedule}\langle f, vs\rangle \qquad q' = q +\!\!+ [(f, vs)]}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', h, q'\rangle}$$

Await
$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{await}\langle v, \rho\rangle \qquad (lc_r, c_a) = \mathrm{L}.\mathsf{splitReturn}(lc', v)}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc_r, h, q +\!\!+ [(lc_a, \rho)]\rangle}$$

Environment Dispatch
$$\frac{[f_i \mid_0^n] = \mathcal{FH}(h, e) \qquad q' = [(f_i, vs) \mid_{i=0}^n]}{\langle lc, h, q\rangle \rightsquigarrow^{\mathsf{fire}\langle e, vs\rangle}_{\mathsf{E}} \langle lc, h, q +\!\!+ q'\rangle}$$

Continuation-Success
$$\frac{\mathrm{L}.\mathsf{final}(lc) \quad q = \kappa : q'}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle \mathcal{CW}_{\mathrm{L}}(lc, \kappa), h, q'\rangle}$$

Continuation-Failure
$$\frac{\mathrm{L}.\mathsf{final}(lc) \quad q = (lc', \rho) : q' \quad \rho(lc) = \mathsf{False}}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc, h, q' +\!\!+ [\kappa]\rangle}$$

Figure 4.7.: Concrete E-semantics: $\langle lc, h, q\rangle \rightsquigarrow^{\epsilon\alpha}_{\mathsf{E}} \langle lc', h', q'\rangle$

[Language Transition] The L-semantics generates the event primitive $\cdot$; the E-semantics leaves the handler register and continuation queue unchanged.

[Add Handler] The L-semantics generates the event primitive $\mathsf{addHdlr}\langle e, f\rangle$; the E-semantics registers the handler f for the event e in the handler register h.

[Remove Handler] The L-semantics generates the event primitive $\mathsf{remHdlr}\langle e, f\rangle$; the E-semantics de-registers the handler f for the event e in the handler register h.

[Synchronous Dispatch] When the L-semantics generates the event primitive $\mathsf{sDispatch}\langle e, vs\rangle$, the E-semantics first creates a handler-continuation for each handler associated with e, together with a yield continuation, $(lc', (\lambda lc.\mathsf{True}))$. These continuations are then all added to the *front* of the continuation queue, ensuring that the handlers will be executed in order, after which the current computation will be retaken unconditionally, given [CW-Yield-Cont.]. Lastly, the E-semantics uses the $\mathsf{suspend}(lc')$ function of the L-semantics, which returns the configuration that is the same as lc' but marked as final, to construct a final configuration lc'', which, given [Continuation-Success], means that the execution of lc' will stop and the first handler will be executed next.

[Asynchronous Dispatch] When the L-semantics generates the event primitive $\mathsf{aDispatch}\langle e, vs\rangle$, the E-semantics proceeds similarly to [Synchronous Dispatch], but the continuations are added to the *back* of the continuation queue rather than to the front, meaning that the handlers will still be executed in order, but at some point in the future.

[Schedule] The L-semantics generates the event primitive $\mathsf{schedule}\langle f, vs\rangle$; the E-semantics creates a handler-continuation (f, vs) for the given function with the given arguments and places it at the *back* of the continuation queue.

[Await] When the L-semantics generates the event primitive $\mathsf{await}\langle v, \rho \rangle$, the E-semantics creates the return configuration, lc_r, and the await configuration, lc_a via the splitReturn function of the L-semantics interface, which constructs: lc_r from lc by setting up the control flow component as if the currently executing function, f, returned the value v; and lc_a from lc by setting up the control flow component to only contain the remainder of the execution of f. It then schedules the remainder of the computation of the currently executing function to be completed asynchronously once ρ holds, and continues the current computation as if the currently executing function had returned the value v.

The remaining three transitions do not rely on the L-semantics. In the [Environment Dispatch] case, the environment generates the event primitive $\mathsf{fire}\langle e, vs \rangle$, and the E-semantics behaves as for [Asynchronous Dispatch], except that the resulting L-configuration does not change. If the current active configuration is final (as checked by the $\mathsf{final}(lc)$ function of the L-semantics interface, which returns `true` if lc is final, and `false` otherwise), the E-semantics tries to create a new configuration for the execution of the continuation at the front of the continuation queue. If this is possible, the execution proceeds ([Continuation-Success]); otherwise, that continuation is demoted to the back of the continuation queue ([Continuation-Failure]).

4.6. Symbolic E-semantics

Symbolic execution [10, 14, 15] is a program analysis technique that systematically explores all possible executions of the given program up to a bound, by executing the program on symbolic values instead of concrete ones. For each execution path, symbolic execution constructs a first-order quantifier-free formula, called a *path condition*, which accumulates the constraints on the symbolic inputs that direct the execution along that path. Here, we describe a symbolic version of the E-semantics introduced in §4.5, obtained by lifting the concrete event semantics to the symbolic level, following well-established approaches [126, 125, 33].

We assume that L has a symbolic semantics with symbolic values, $\hat{v} \in \widehat{\mathcal{V}}$, built using symbolic variables, $\hat{x} \in \hat{\mathcal{X}}$. The concepts given in Figure 4.4, but for symbolic instead of concrete values, and are annotated with ˆ to be distinguishable from their concrete counterparts; for example, we have: *symbolic events*, $\hat{e} \in \widehat{\mathcal{E}} \subset \widehat{\mathcal{V}}$; *symbolic handler registers*, $\hat{h} \in \widehat{\mathcal{H}} : \widehat{\mathcal{E}} \rightharpoonup \overline{\mathcal{F}}$, mapping symbolic events to lists of function identifiers; and symbolic E-configurations, $\widehat{ec} \in \widehat{\mathcal{EC}}$, comprising a symbolic L-configuration, $\widehat{lc} \in \widehat{\mathcal{LC}}$, a *symbolic handler register*, and a *symbolic continuation queue*, $\hat{q} \in \widehat{\mathcal{Q}}$, which is a list of *symbolic continuations*, $\hat{\kappa} \in \widehat{\mathcal{K}}$. We also assume that every symbolic L-configuration $\widehat{lc}$ has a way to record a boolean symbolic value, $\pi \in \Pi \subset \widehat{\mathcal{V}}$, to which we refer as the *path condition* of $\widehat{lc}$.

The symbolic E-semantics, like the concrete, relies on a L-semantics interface, which is an extended version of the one defined in §4.5.1 to handle symbolic execution. To allow for the branching during symbolic execution, we generalise auxiliary functions of the E-semantics. Consequently, they are defined as relations in the symbolic E-semantics. In the following, we introduce the L-semantics interface needed by the symbolic E-semantics (§4.6.1) and the auxiliary relations of the symbolic E-semantics (§4.6.2). Then, we present the rules of our symbolic E-semantics (§4.6.3) and correctness results (§4.6.4).

4.6.1. Symbolic L-semantics Interface

In addition to the auxiliary functions provided by the concrete L-semantics, the symbolic E-semantics relies on the **assume** and $\mathsf{pc}()$ functions in order to compute the path condition during symbolic execution. We describe the two functions below.

1. $\mathsf{assume}(\widehat{lc}, \pi) = \widehat{lc}'$, where $\widehat{lc}'$ is obtained from $\widehat{lc}$ by extending its path condition with the formula π, if such an extension is satisfiable

2. $\mathsf{pc}(\widehat{lc}) = \pi$, where π is the path condition computed in the current branch of configuration $\widehat{lc}$. We leave the computation of the path condition to the underlying language L, meaning that $\mathsf{pc}(\langle \widehat{lc}, \hat{h}, \hat{q} \rangle) = \mathsf{pc}(\widehat{lc})$

4.6.2. Auxiliary Relations of the Symbolic E-semantics

Analogously to the concrete E-semantics, the symbolic E-semantics relies on auxiliary functions for modularity and readability purposes. Those auxiliary functions introduced for the concrete E-semantics that do not operate on handler registers are also applicable for the symbolic E-semantics. The ones that operate on handler registers ($\mathcal{AH}$, $\mathcal{RH}$, and $\mathcal{FH}$) need to branch as the symbolic E-semantics assumes that events can be symbolic values. To account for this branching, we pair each outcome with a constraint describing the conditions under which the outcome is valid. For each operation, we branch on two possible scenarios: **(1)** either assuming that the event $\hat{e}$ is found on the domain of the handler register, in which case we generate the constraint $\hat{e} = \hat{e}'$ or **(2)** assuming that the event $\hat{e}$ is not found on the domain of the handler register, in which case we generate the constraint $\hat{e} \notin \mathtt{dom}(\hat{h})$. The auxiliary function Continue With (CW) is defined analogously to its concrete counterpart, as in the two possible cases of continuations, it simply calls functions provided by the underlying language interface. In Figure 4.8, we define the auxiliary relations of the symbolic E-semantics.

Add Handler - Found

$$\frac{\hat{e}' \in \mathtt{dom}(\hat{h}) \qquad \hat{h}' = \hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') +\!\!+ [f]\right]}{\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \hat{e} = \hat{e}')}$$

Add Handler - Not Found

$$\frac{\hat{h}' = \hat{h}\,[\hat{e} \mapsto [f]]}{\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \hat{e} \notin \mathtt{dom}(\hat{h}))}$$

Remove Handler - Found

$$\frac{\hat{e}' \in \mathtt{dom}(\hat{h}) \qquad \hat{h}' = \hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') \setminus f\right]}{\mathcal{RH}((\hat{h}, \hat{e}, f)) \rightsquigarrow ((\hat{h}', \hat{e} = \hat{e}'))}$$

Remove Handler - Not Found

$$\mathcal{RH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}, \hat{e} \notin \mathtt{dom}(\hat{h}))$$

Find Handler - Found

$$\frac{\hat{e}' \in \mathtt{dom}(\hat{h})}{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow (\hat{h}(\hat{e}'), \hat{e} = \hat{e}')}$$

Find Handler - Not Found

$$\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([], \hat{e} \notin \mathtt{dom}(\hat{h}))$$

CW-Handler-Cont.

$$\mathcal{CW}_{\mathrm{L}}(\widehat{lc}, (f, \hat{vs})) \triangleq \mathsf{L.initialConf}(\widehat{lc}, (f, \hat{vs}))$$

CW-Yield-Cont.

$$\frac{\rho(\widehat{lc}) = \mathsf{True}}{\mathcal{CW}_{\mathrm{L}}(\widehat{lc}, (\widehat{lc}', \rho)) \triangleq \mathsf{L.mergeConfs}(\widehat{lc}, \widehat{lc}')}$$

Figure 4.8.: Symbolic E-semantics: Auxiliary Relations

4.6.3. Symbolic E-semantics Rules

An excerpt of the symbolic E-semantics is given in Figure 4.9. We focus on the representative rules that are different from their concrete counterparts, highlighting the differences in grey. These differences are introduced by the above-discussed branching of the auxiliary relations. In particular, every time an auxiliary relation is used, the constraint it generates must be added to the current path condition using the $\mathsf{assume}(\widehat{lc}, \pi)$ function of the L-semantics interface, which returns the symbolic L-configuration obtained by extending the path condition of $\widehat{lc}$ with the formula π if such an extension is satisfiable, and is undefined otherwise.

Add Handler

$$\frac{\widehat{lc} \leadsto^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{addHdlr}\langle \hat{e}, f\rangle \qquad \mathcal{AH}(\hat{h}, \hat{e}, f) \leadsto (\hat{h}', \pi) \quad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \leadsto_{\hat{\mathsf{E}}} \langle \widehat{lc}'', \hat{h}', \hat{q}\rangle}$$

Environment Dispatch

$$\frac{\mathcal{FH}(\hat{h}, \hat{e}) \leadsto ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, \hat{vs}) \mid_{i=0}^n] \qquad \widehat{lc}' = \mathrm{L}.\mathsf{assume}(\widehat{lc}, \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \leadsto_{\hat{\mathsf{E}}}^{(\mathsf{fire}\langle \hat{e}, \hat{vs}\rangle} \langle \widehat{lc}', \hat{h}, \hat{q} +\!\!+ \hat{q}'\rangle}$$

Synchronous Dispatch

$$\frac{\widehat{lc} \leadsto^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{sDispatch}\langle \hat{e}, \hat{vs}\rangle \qquad \mathcal{FH}(\hat{h}, \hat{e}) \leadsto ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, \hat{vs}) \mid_{i=0}^n] \qquad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi) \quad \widehat{lc}''' = \mathrm{L}.\mathsf{suspend}(\widehat{lc}'')}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \leadsto_{\hat{\mathsf{E}}} \langle \widehat{lc}''', \hat{h}, \hat{q}' +\!\!+ [(\widehat{lc}'', (\lambda \widehat{lc}.\mathsf{True}))] +\!\!+ \hat{q}\rangle}$$

Asynchronous Dispatch

$$\frac{\widehat{lc} \leadsto^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{aDispatch}\langle \hat{e}, \hat{vs}\rangle \qquad \mathcal{FH}(\hat{h}, \hat{e}) \leadsto ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, \hat{vs}) \mid_{i=0}^n] \qquad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \leadsto_{\hat{\mathsf{E}}} \langle \widehat{lc}'', \hat{h}, \hat{q} +\!\!+ \hat{q}'\rangle}$$

Figure 4.9.: Symbolic E-semantics (excerpt): $\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \leadsto_{\hat{\mathsf{E}}}^{\hat{\epsilon\alpha}} \langle \widehat{lc}', \hat{h}', \hat{q}'\rangle$

4.6.4. Correctness

A symbolic E-semantics is correct w.r.t. a concrete E-semantics if it satisfies the following properties: *Directed Soundness*, which holds when every symbolic trace over-approximates all concrete traces that follow its execution path; and *Directed Completeness*, which holds when every symbolic trace has at least one valid concretisation. Note that, because the E-semantics relies on an underlying L-semantics, for the E-semantics to satisfy these two properties, the underlying L-semantics also needs to satisfy them. If a symbolic E-semantics is correct, we are able to guarantee, for instance, the absence of false-positive bug-reports: if a bug happens symbolically, then it must also happen concretely.

To establish the correctness of the symbolic E-semantics w.r.t the concrete E-semantics, we first relate the corresponding E-configurations using *symbolic environments*, $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$, which map symbolic variables to concrete values, while preserving types. Given a symbolic environment ε, we write $\mathcal{I}_\varepsilon(\hat{v})$ to denote the interpretation of $\hat{v}$ under ε, with the key case being that of symbolic variables: $\mathcal{I}_\varepsilon(\hat{x}) = \varepsilon(\hat{x})$. We extend $\mathcal{I}_\varepsilon$ to all other concepts defined in Figure 4.4 component-wise, overloading notation, and provide the corresponding definitions in Figure 4.10. For example, $\mathcal{I}_\varepsilon(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle) \triangleq \langle \mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q})\rangle$. We assume that interpretation is preserved by the functions of the L-semantics interface; for example, that $\mathrm{L}.\mathsf{final}(\widehat{lc}) \Leftrightarrow \mathrm{L}.\mathsf{final}(\mathcal{I}_\varepsilon(\widehat{lc}))$.

We define the *models* of a symbolic L-configuration $\widehat{lc}$ under the path condition π as the set of all concrete configurations obtained via interpretations of $\widehat{lc}$ that satisfy π and their accompanying symbolic environments: $\mathcal{M}_\pi(\widehat{lc}) = \{(\varepsilon, \mathcal{I}_\varepsilon(\widehat{lc})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$. We extend this notion to symbolic event primitives, environment actions, and E-configurations, overloading notation.

HR - Empty
$\mathcal{I}_\varepsilon(\emptyset) \triangleq \emptyset$

HR - Composition
$\mathcal{I}_\varepsilon(\hat{h}_1 \uplus \hat{h}_2) \triangleq \mathcal{I}_\varepsilon(\hat{h}_1) \uplus \mathcal{I}_\varepsilon(\hat{h}_2)$

HR - Cell
$\mathcal{I}_\varepsilon([\hat{e} \mapsto \overline{f}]) \triangleq [\mathcal{I}_\varepsilon(\hat{e}) \mapsto \overline{f}]$

CQ - Empty
$\mathcal{I}_\varepsilon([]) \triangleq []$

CQ - Non-Empty
$\mathcal{I}_\varepsilon(\hat{\kappa} : \hat{q}) \triangleq \mathcal{I}_\varepsilon(\hat{\kappa}) : \mathcal{I}_\varepsilon(\hat{q})$

Cont - Hanlder-Cont
$\mathcal{I}_\varepsilon(f, [\hat{v}_1, ..., \hat{v}_n]) \triangleq (f, [\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n)])$

Cont - Yield-Cont
$\mathcal{I}_\varepsilon(\widehat{lc}, \hat{\rho}) \triangleq (\mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{\rho}))$

Event Primitive - AH/RH
$$\frac{\mathsf{prim} \in \{\mathsf{addHdlr}, \mathsf{remHdlr}\}}{\mathcal{I}_\varepsilon(\mathsf{prim}\langle \hat{e}, f\rangle) \triangleq \mathsf{prim}\langle \mathcal{I}_\varepsilon(\hat{e}), f\rangle}$$

Event Primitive - SD/AD
$$\frac{\mathsf{prim} \in \{\mathsf{sDispatch}, \mathsf{aDispatch}\}}{\mathcal{I}_\varepsilon(\mathsf{prim}\langle \hat{e}, \hat{vs}\rangle) \triangleq \mathsf{prim}\langle \mathcal{I}_\varepsilon(\hat{e}), \mathcal{I}_\varepsilon(\hat{vs})\rangle}$$

Event Primitive - Schedule
$\mathcal{I}_\varepsilon(\mathsf{schedule}\langle f, [\hat{v}_1, ..., \hat{v}_n]\rangle) \triangleq \mathsf{schedule}\langle f, [\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n)]\rangle$

Event Primitive - Await
$\mathcal{I}_\varepsilon(\mathsf{await}\langle \hat{\rho}\rangle) \triangleq \mathsf{await}\langle \mathcal{I}_\varepsilon(\hat{\rho})\rangle$

EAction - Event
$\mathcal{I}_\varepsilon((\mathsf{fire}\langle \hat{e}, [\hat{v}_1, ..., \hat{v}_n]\rangle) \triangleq (\mathsf{fire}\langle \mathcal{I}_\varepsilon(\hat{e}), [\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n)]\rangle)$

EAction - UL
$\mathcal{I}_\varepsilon(\cdot) \triangleq \cdot$

E-semantics Configuration
$\mathcal{I}_\varepsilon(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle) \triangleq \langle \mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q})\rangle$

Figure 4.10.: Interpretation of E-semantics Structures

The correctness of the E-semantics relies on the correctness of the L-semantics. The notion of correctness for the L-semantics is analogous to the E-semantics: as formalised in Definition 4.1, a given symbolic L-semantics is correct w.r.t. a given concrete L-semantics if every symbolic trace: **(1)** over-approximates all concrete traces that follow its execution path and whose initial concrete L-configuration is over-approximated by the initial symbolic L-configuration (*Directed Soundness*); and **(2)** has at least one concretisation (*Directed Completeness*).

Definition 4.1 (Correctness Criteria - Symbolic L-Semantics)**.**

L-Directed-Soundness
$$\begin{array}{l}\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{lc}')) \\ \quad \wedge (\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc}) \wedge lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \\ \qquad \implies (\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})\end{array}$$

L-Directed-Completeness
$$\begin{array}{l}\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{lc}')) \\ \quad \wedge (\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc}) \\ \qquad \implies \exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'\end{array}$$

Because the L-semantics is opaque to the E-semantics, in order to prove the correctness of the E-semantics, we assume the correctness of the L-semantics. We instantiate our E-semantics with JSIL in JaVerT.Click. We previously proved [35] a notion correctness which is analogous to the one formalised by Definition 4.1. Our results guarantee that the symbolic semantics of JSIL is correct with respect to its concrete counterpart. When instantiating the E-semantics with a different language, one needs to guarantee that the language semantics satisfy both directed soundness and directed completeness.

Finally, we formalise the correctness of the symbolic E-semantics w.r.t. the concrete E-semantics in Theorem 4.1, which states that if the symbolic L-semantics is correct, then so is the obtained E-semantics. To precisely identify the concrete traces that follow the same path as the symbolic trace, we only pick concretisations of the initial symbolic state that satisfy the *final* path condition $(\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}'))$.

Theorem 4.1 (Correctness of the Symbolic E-semantics).

E-Directed-Soundness

$$\widehat{\epsilon c} \rightsquigarrow_{\hat{\mathsf{E}}}^{\hat{\epsilon\alpha}} \widehat{\epsilon c}' \wedge \pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}') \wedge (\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \wedge (\varepsilon, \epsilon\alpha) \in \mathcal{M}_\pi(\hat{\epsilon\alpha}) \wedge \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c' \implies (\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}')$$

E-Directed-Completeness

$$\widehat{\epsilon c} \rightsquigarrow_{\hat{\mathsf{E}}}^{\hat{\epsilon\alpha}} \widehat{\epsilon c}' \wedge \pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}') \wedge (\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \implies \exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$$

The proof is done by case analysis on the symbolic rules for the E-semantics, and can be found integrally in the appendix (§A).

5. Message-Passing Semantics

The WebWorkers API [133] was added to the HTML5 standard [138] to enable the use of multiple threads in JavaScript Web programs, which, until their introduction, executed on a single thread. With this API, client-side JavaScript programs can execute time-consuming operations in the background without blocking the main browser thread. Web Workers run concurrently, have their own memory, and communicate via messages through the use of the WebMessaging API [140], which follows the message-passing concurrency paradigm [19, 65, 69]. Besides introducing their own complex features, these APIs depend on an underlying event model, as they both make use of DOM Events [139]. To support the symbolic analysis of the WebMessaging and WebWorkers APIs, we design a Message-passing Semantics which is expressive enough to capture the multi-threaded nature of the WebWorkers API and, at the same time, the message-passing communication model of the WebMessaging API.

Our message-passing semantics (onward: MP-semantics) formalises the message-passing model of the WebMessaging and WebWorkers APIs, and, to the best of our knowledge, is the first formal model targeting the WebMessaging and WebWorkers APIs. There are previous works [119, 121] on program analyses targeting the WebMessaging API, but they are only focussed on finding security vulnerabilities and cannot provide bounded correctness guarantees. Furthermore, they do not handle the multithreaded nature of the WebWorkers API. Our tool is the first to analyse Web programs calling WebWorkers. Additionally, JaVerT.Click, comes with a symbolic execution engine that allows us to prove the bounded correctness of functional properties and find bugs in JavaScript programs calling multiple Web APIs, including the DOM [136], WebMessaging and WebWorkers.

Outline. We first provide a motivating example making use of the WebMessaging and WebWorkers APIs (§5.1). As the message-passing model of these two APIs rely on DOM Events, we build our MP-semantics parametrically on an underlying event semantics. We then explain the parametric construction of the MP-semantics (§5.2) and introduce our message-passing syntax (§5.3). Next, we show how JaVerT.Click performs symbolic analysis using the motivating example (§5.4). Although our MP-semantics is general, for clarity purposes, we introduce the MP-semantics by first assuming its instantiation with a concrete E-semantics (§5.5) and finally assuming its instantiation with a symbolic E-semantics (§5.6) MP-semantics.

5.1. Motivating Example

We go through a simple program that uses the WebMessaging and WebWorkers APIs, showing how JaVerT.Click can be used to identify bugs in real world message-passing Web programs. In particular, we consider a client program of the `webworker-promise` library [105], which we later use to evaluate our tool. This library is an open-source library that functions as a wrapper around the WebMessaging and WebWorkers APIs, providing extra functionality mainly related to the use of JavaScript promises.

Web workers communicate via messages through the use of the WebMessaging API. The `webworker-promise` library allows developers to create worker-promises, which represent the result of the computation performed by a worker. Analogously to promises, the computation associated with a worker-promise is executed asynchronously. When such a computation is completed, the respective worker-promise transitions from the *pending* state to either the *fulfilled* or *rejected* state. More concretely, the computation associated with a worker-promise is executed by a new worker created specifically to that effect. When the computation is completed, the worker thread sends a message with the result to the main thread, causing the associated promise to be either fulfilled or rejected.

Below, we give a simple symbolic test for the `webworker-promise` library, which uncovered a bug in the library code. This test guarantees the bounded correctness of the following functional property:

> *If the main thread sends a message to the worker thread and the worker thread sends this same message back to the main thread, the message received in the main thread is equal to the one that was sent to the worker thread.*

The test is composed of a worker script, capturing the asynchronous computation, and a main script, which is charge of creating the worker-promise and checking its result.

Worker script. The implementation of the worker script is given in Figure 5.1. This worker receives messages and returns them to the main thread. In line 1, we import the `registerWebworker` function from the `webworker-promise` library. In lines 2-4, we call `registerWebworker` with a handler as argument that will be executed whenever a message arrives to the worker. It is the job of the `registerWebworker` function to send the value returned by the handler (in this case, the value stored in the `message` variable) back to the main thread.

```
1 const registerWebworker = require('webworker-promise/lib/register');
2 registerWebworker((message) => {
3   return message;
4 });
```

Figure 5.1.: Worker script (`worker.js`)

Main script. We give the implementation of the main script in Figure 5.2, highlighting the usage of the WebWorkers API in brown. In line 1, we import the `WebworkerPromise` constructor from the `webworker-promise` library. In line 2, we declare the `worker` variable, representing a worker-promise object created by calling the `WebworkerPromise` constructor, which takes a worker object as input. We create the worker by providing `worker.js` as filename, so that the main script can be executed in parallel with the worker script given in Figure 5.1. Next, in line 3, we declare the variable `msg` and assign it a fresh symbolic value using the `symb`() primitive provided by JaVerT.Click. In line 4, we constrain the values that `msg` can represent by stating that it must be of type `object`. Next, in line 6, we send the message to the worker by calling `postMessage` on the `worker` object, with this call simply returning a promise object representing the value the worker will eventually send back to the main thread. In lines 7-9, we define a `then` handler to be triggered when a message arrives back to the main thread from the worker. As the worker defined in Figure 5.1 is supposed to only mirror the messages it receives, we check, in line 8, that the content of the incoming message, `response`, is equal to the content of the outgoing message `msg` (by calling `assertDeepEqual(response, msg)`). Finally, in lines 10-12, we define a `catch` handler to be triggered if there is an error in the communication process between the main and worker threads. As this is not expected to happen, we add an `assert(false)` to the body of the error handler.

```
1  const WebworkerPromise = require('webworker-promise');
2  var worker = new WebworkerPromise(new Worker('worker.js'));
3  var msg = symb();
4  assume(typeof msg === 'object')
5
6  worker.postMessage(msg)
7  .then((response) => {
8      assertDeepEqual(response, msg);
9  })
10 .catch(err => {
11     assert(false); // This should not happen!
12 });
```

Figure 5.2.: Main script

Bug found. By running this test using the symbolic testing engine of JaVerT.Click we identified a bug in the library. The message `msg` is sent successfully from the main thread to the worker thread. However, the message does not arrive successfully back to the main thread. More precisely, there is at least a concrete value for the symbolic variable `msg` that triggers the `catch` clause instead of the `then` clause. JaVerT.Click reports the failing model [`msg` : `null`], meaning that when `msg` has value `null`, an error occurs during the communication from the worker back to the main thread.

The bug is caused by a type error in the snippet of the library code shown in Figure 5.3. The function `isPromise`, defined in line 1, is used to check whether a given object `o` is a promise. In line 4, `isPromise` is called with the result given by the worker to be sent back to the main thread. As the worker just mirrors the messages sent from the main thread, if the message is `null`, the variable `result` will also be `null`, causing the execution of `isPromise` to raise a TypeError. In JavaScript, the value `null` has type `object`, causing the first conjunct of `isPromise` to evaluate to true and, subsequently, triggering a null-pointer dereference in the evaluation of the second conjunct. Consequently, the communication fails and so does the test.

```
1  const isPromise = o => typeof o === 'object' &&
2    typeof o.then === 'function' && typeof o.catch === 'function';
3   ...
4   if(isPromise(result)) {
5      ...
6  }
7  ...
```

Figure 5.3.: Library code (register.js file)

We found the bug described above while testing the `webworker-promise` library using JaVerT.Click.

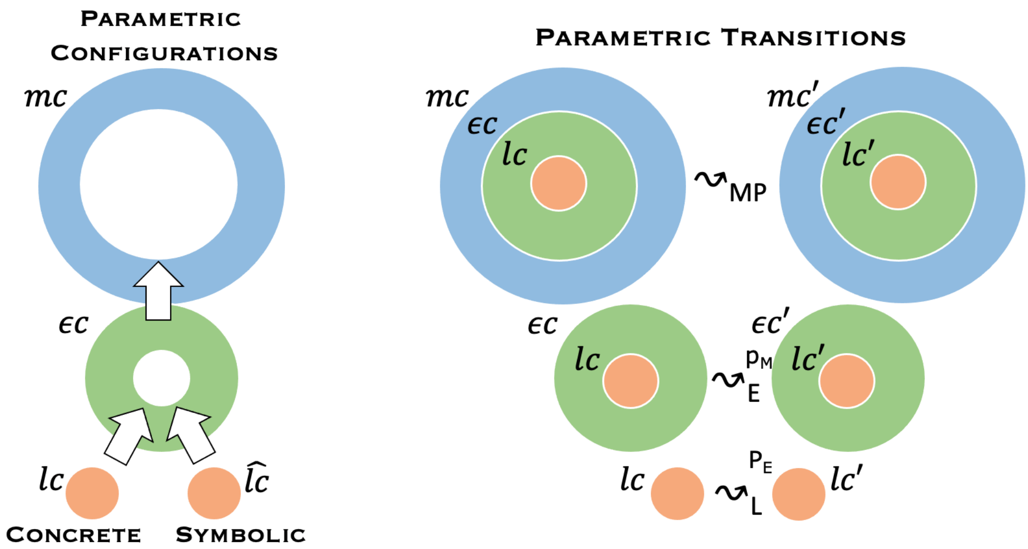

Figure 5.4.: The parametric construction of the MP-semantics

The bug was reported[1] to library's developers and fixed via a pull request.[2] The fix consisted of adding the `o! == null` check immediately before the `o.then` property access in the definition of `isPromise`.

This kind of bug can be hard to detect without the use of a symbolic execution engine like the one provided by JaVerT.Click. The `webworker-promise` library does contain a concrete test suite. The developers, however, did not test this specific case and the bug went unnoticed. While there are other symbolic execution tools for JavaScript [109, 116, 84], none of them provides support for the WebMessaging and WebWorkers APIs. Our tool is the first to implement a symbolic execution engine that analyses message-passing JavaScript programs calling the WebMessaging and WebWorkers APIs.

5.2. Parametric Construction

Our MP-semantics is parametric on: a scheduler, responsible for choosing which thread executes at each computation step; and an event semantics (a.k.a. E-semantics, introduced in Chapter 4), which is itself parametric on a language semantics (a.k.a. L-semantics). The MP-semantics interacts with the E-semantics via *message-passing primitives*, $\mathrm{p} \in \mathrm{P}$, which capture the essence of the fundamental operations underpinning the targeted APIs such as sending messages and creating workers. In Figure 5.4, we show how our MP-semantics is parametric on the underlying E-semantics both in terms of configurations (left) and transitions (right).

Parametric Configurations. Every MP-configuration mc computes a sequence of event configurations, each representing an active thread. This means that an MP-configuration mc is parametric on an E-configuration ϵc. Each E-configuration ϵc is parametric on a language configuration, which can be either concrete (lc) or symbolic ($\widehat{lc}$). Hence, both the E-semantics and the MP-semantics are generic in the sense that they can be instantiated with either a concrete semantics or a symbolic semantics.

[1] Issue: `https://github.com/kwolfy/webworker-promise/issues/9`
[2] Pull Request: `https://github.com/kwolfy/webworker-promise/pull/11`

Parametric Transitions. An MP-transition has the form $mc \rightsquigarrow_{\mathsf{MP}} mc'$, where mc and mc' are MP-configurations. Because the MP-semantics is parametric on an E-semantics, which is itself parametric on an L-semantics, if the L-semantics makes a step, the E-semantics follows accordingly, and so does the MP-semantics. Each E-transition and L-transition have the form $\epsilon c \rightsquigarrow_{\mathsf{E}}^{\mathrm{p_M}} \epsilon c'$ and $lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p_E}} lc'$, respectively, and can be concrete or symbolic. The L-semantics generates an event primitive $\mathrm{p}_{\mathtt{E}}$, which tells the E-semantics if the current command is event-related (meaning that it requires, for instance, associating an event to a given handler or dispatching an event). The E-semantics may take action to process the corresponding event-related action and then generates a message-passing primitive $\mathrm{p}_{\mathtt{M}}$ which tells the MP-semantics if the current command involves message-passing (meaning that it requires, for instance, sending a message from one worker to another). Finally, the MP-semantics processes the message-passing primitive $\mathrm{p}_{\mathtt{M}}$.

5.3. Message-Passing Syntax

We introduce our message-passing syntax in Figure 5.5, highlighting in blue the elements provided by the E-semantics and green the elements provided by the MP-semantics. The MP-semantics inherits the values $v \in \mathcal{V}$ and variables $x \in \mathcal{X}$ of the underlying language semantics. These values can be either concrete or symbolic. An event configuration (onward: configuration) $\epsilon c \in \mathcal{EC}$ is opaque to the MP-semantics. An MP-configuration consists of a sequence of event configurations, where each configuration represents a running thread and is assigned an identifier $\alpha \in \mathcal{A}$.

Running threads communicate via ports $p \in \mathcal{P}$, which are analogous to those defined in the WebMessaging API [140]. A port can be *connected* to a set of other ports, allowing messages to be sent from the configuration that owns the port to the configurations of the ports to which it is connected. Accordingly, a message $m \in \mathcal{M}$ is a pair (vs, ps) consisting of a value list vs and a port list ps, respectively corresponding to the data being sent and the ports being *transferred*. The WebMessaging standard allows for transferable objects[3] to be sent from one configuration to another. Hence, configurations can communicate effectively by transferring, for instance, array buffer objects. `MessagePort` objects are also transferable objects, which means that ports can be sent among threads. Effectively, if a port is transferred, its messages are redirected and it becomes inaccessible from its origin after it is transferred. For instance, consider that we have a main thread and two worker threads `w1` and `w2`. If a port is transferred from main to `w1`, messages previously sent to that port need to be delivered to `w1` instead of the main thread.

The MP-semantics and the E-semantics communicate through primitives $\mathrm{p} \in \mathrm{P}$, which capture the essence of the fundamental message-passing operations required to model targeted APIs. We use $\cdot$ meaning that no operation is required; the primitive $\mathsf{send}\langle vs, ps, p_1, p_2\rangle$ is used for sending a message consisting of the pair containing vs and ps from port p_1 to port p_2. The primitive $\mathsf{create}\langle x, vs\rangle$ is used to create a new worker, which is modelled as a configuration to launch a new worker, resulting in the creation of a new configuration. The variable x is assigned to a freshly generated unique identifier α that is associated with the newly created configuration. The value list vs should contain the information necessary to setup the new configuration, which could include, for instance, a path to the file with the code of the thread to be executed. This behaviour is analogous to the one

[3] `https://html.spec.whatwg.org/multipage/structured-data.html#transferable-objects`

Values	**Variables**	**E-confs**	**E-conf Ids**	**Ports**	**Messages**
$v \in \mathcal{V}$	$x \in \mathcal{X}$	$\epsilon c \in \mathcal{EC}$	$\alpha \in \mathcal{A} \subset \text{Int}$	$p \in \mathcal{P} \subset \text{Int}$	$m \in \mathcal{M} := (vs, ps)$

Message-passing Primitives

$\mathrm{p} \in \mathrm{P} := \cdot \mid \mathsf{send}\langle vs, ps, p_1, p_2\rangle \mid \mathsf{create}\langle x, vs\rangle \mid \mathsf{terminate}\langle \alpha\rangle \mid \mathsf{newPort}\langle\rangle \mid \mathsf{connect}\langle p_1, p_2\rangle \mid \mathsf{disconnect}\langle p\rangle \mid \mathsf{getConnected}\langle x, p\rangle \mid \mathsf{notifyAll}\langle v, vs\rangle \mid \mathsf{fire}\langle v, vs\rangle \mid \mathsf{beginAtomic} \mid \mathsf{endAtomic}$

E-Conf Sequences	**Message Queues**	**Port-confs Map**	**Conn-ports Map**
$cs \in \mathcal{CS} : \overline{\mathcal{EC} \times \mathcal{A}}$	$mq \in \mathcal{MQ} : \overline{\mathcal{M} \times \mathcal{P}}$	$pcm \in \mathcal{PCM} : \mathcal{P} \rightharpoonup \mathcal{A}$	$cpm \in \mathcal{CPM} : \mathcal{P} \rightharpoonup \overline{\mathcal{P}}$

Lead Confs	**MP-Configurations**
$\ell \in \mathcal{L} := \cdot \mid \mathtt{Conf}\langle \alpha \rangle$	$mc \in \mathcal{MC} : \mathcal{CS} \times \mathcal{MQ} \times \mathcal{PCM} \times \mathcal{CPM} \times \mathcal{L}$

Configuration Actions

$ca \in \mathcal{CA} := \cdot \mid \mathtt{Add}\langle \epsilon c, \alpha\rangle \mid \mathtt{Rem}\langle \alpha\rangle \mid \mathtt{Hold}\langle \alpha\rangle \mid \mathsf{Free}\langle \alpha\rangle \mid \mathsf{Notify}\langle v, vs\rangle$

Figure 5.5.: Message-Passing Syntax

described by the WebWorkers API [133]. Configurations can be terminated immediately through $\mathsf{terminate}\langle\alpha\rangle$. The primitives $\mathsf{newPort}\langle\rangle$, $\mathsf{connect}\langle p_1, p_2\rangle$, $\mathsf{disconnect}\langle p\rangle$ and $\mathsf{getConnected}\langle x, p\rangle$ are used to create a fresh port in the current configuration (different from all existing ports), connect two ports, disconnect all ports connected with a given port, and obtain all ports connected with the one given as input. The primitive $\mathsf{notifyAll}\langle v, vs\rangle$ allows the currently active configuration to trigger an event on all configurations. The value v represents the event to be triggered in all configurations and vs the list of arguments to be given to the event handlers associated with v. The primitive $\mathsf{fire}\langle v, vs\rangle$ is used for firing an event v with arguments vs using the E-semantics.

The primitives beginAtomic and endAtomic allow the execution of atomic blocks in a specific configuration. This is essential to avoid data races between configurations. In order to prevent the interleaving between atomic blocks and other code, we make use of lead configurations $\ell \in \mathcal{L}$. Essentially, if a configuration enters an atomic block, it becomes a lead configuration and the MP-semantics is forced to prioritise its instructions until it finishes executing the atomic block. Formally, a lead configuration is either $\mathtt{Conf}\langle\alpha\rangle$ meaning that the configuration with identifier α should be prioritised or $\cdot$ if no configuration is executing an atomic block. This is useful to impose a certain scheduling policy at the code level. For instance, one could always guarantee that there would be no thread interleaving during the execution of a script. Although most browsers seem to adhere this policy, other implementations could opt for a highly interleaving approach. This will be detailed in §5.5.

As we deal with multiple executions, the MP-semantics maintains a configuration sequence $cs \in \mathcal{CS}$. Each element of the sequence contains a configuration ϵc and its respective identifier α. We use ϵc_α to denote each pair of the configuration sequence. In order to handle pending messages, we make use of a message queue $mq \in \mathcal{MQ}$, containing a list of pairs, each consisting of a message and a destination port. We associate ports with configurations through a *port configuration map* $pcm \in \mathcal{PCM}$, which maps ports to configuration identifiers. The *connected-ports map* $cpm \in \mathcal{CPM}$ keeps track of the connection between ports; for instance, if $cpm(p_1) = [p_2, p_3]$, then messages sent through port p_1 arrive at both p_2 and p_3. In summary, an MP-configuration $mc \in \mathcal{MC}$ is formed by: **(1)** a configuration sequence cs; **(2)** a message queue mq containing all pending messages to be processed; **(3)** a port-configurations map pcm; **(4)** a connected-ports map cpm and **(5)** a lead configuration ℓ.

Finally, the MP-semantics makes use of configuration actions $ca \in \mathcal{CA}$ that allow updating configuration sequences in a two-step fashion. When no action is required, we use the symbol $\cdot$ to signify that the configuration queue is to be left unchanged. A configuration action ca can represent, for instance, the addition of a new configuration c with identifier α, $\texttt{Add}\langle \epsilon c_\alpha \rangle$, or the removal of the configuration with identifier α, $\texttt{Rem}\langle \alpha \rangle$. A configuration action can also denote the beginning, $\texttt{Hold}\langle \alpha \rangle$, and ending, $\mathsf{Free}\langle \alpha \rangle$, of atomic blocks. Finally, an action can also denote the notify action $\mathsf{Notify}\langle v, vs \rangle$, which causes the event represented by the value v to be triggered on all configurations with arguments vs. Configuration actions are explained in detail in §5.5.

5.4. Symbolic Analysis of Motivating Example

Figure 5.6 shows a fragment of the symbolic execution tree resulting from the execution of the motivating example introduced in §5.1. Each transition is labelled with the command that triggered it as well as one of the letters M or W to indicate if the command was issued in the main (M) or worker (W) thread. We also label each MP-configuration with a number. The initial MP-configuration (1) contains: a configuration sequence *cs* with a single configuration ϵc_{M} corresponding to the main thread, an empty message queue *mq*, an empty port-configurations map *pcm* and an empty connected-ports map *cpm*. We do not represent lead configurations as the example does not make direct use of atomic blocks, meaning that the lead e-configuration of the illustration configurations is always $\cdot$.

After the worker is created by the main thread, the following changes are applied to the MP-configuration: **(1)** the E-configuration of the worker thread, ϵc_{W}, is added to the configuration sequence *cs*; **(2)** the port configuration map *pcm* is updated with the ports p1 and p2, which are created to allow for the communication between the main and worker threads. Port p1 belongs to the configuration ϵc_{M} and p2 to ϵc_{W}; **(3)** the connected-ports map *cpm* receives two entries so that ports p1 and p2 are effectively connected with each other. We then obtain MP-configuration 2.

Next, the main thread sends a message to the worker thread via `worker.postMessage(msg)`. As a consequence, a message containing the pair (#msg, p2) is added to the message queue *mq*, meaning that message #msg must be sent to port p2. We then obtain the MP-configuration 3. We use #msg to denote the symbolic variable associated with the program variable `msg`. We use $-||-$ to indicate that the value of the corresponding element in the current state is the same as the one of the previous state.

The worker thread then processes the message sent from the main thread. As explained before (cf. §5.1), the worker thread simply sends back the value it receives to the main thread and this is enabled via `return message`, leading to the MP-configuration 4. Hence, after the message is processed in the worker thread, the symbolic value #msg should be sent back to the main thread. However, the symbolic execution of the function in charge of processing the message branches on the value of the symbolic variable #msg. If #msg has value null, an error occurs in the code of the `webworker-promise` library, as shown in Figure 5.3. In this case, an error is sent back (MP-configuration 5), consequently triggering the test failure in the main thread captured by `assert(false)` (MP-configuration 6). In contrast, if #msg does not have value null, no error occurs during the communication process, and #msg is sent back to the main thread (MP-configuration 7). In this case, the test passes after the successful call to `assertDeepEqual(response, msg)` (MP-configuration 8).

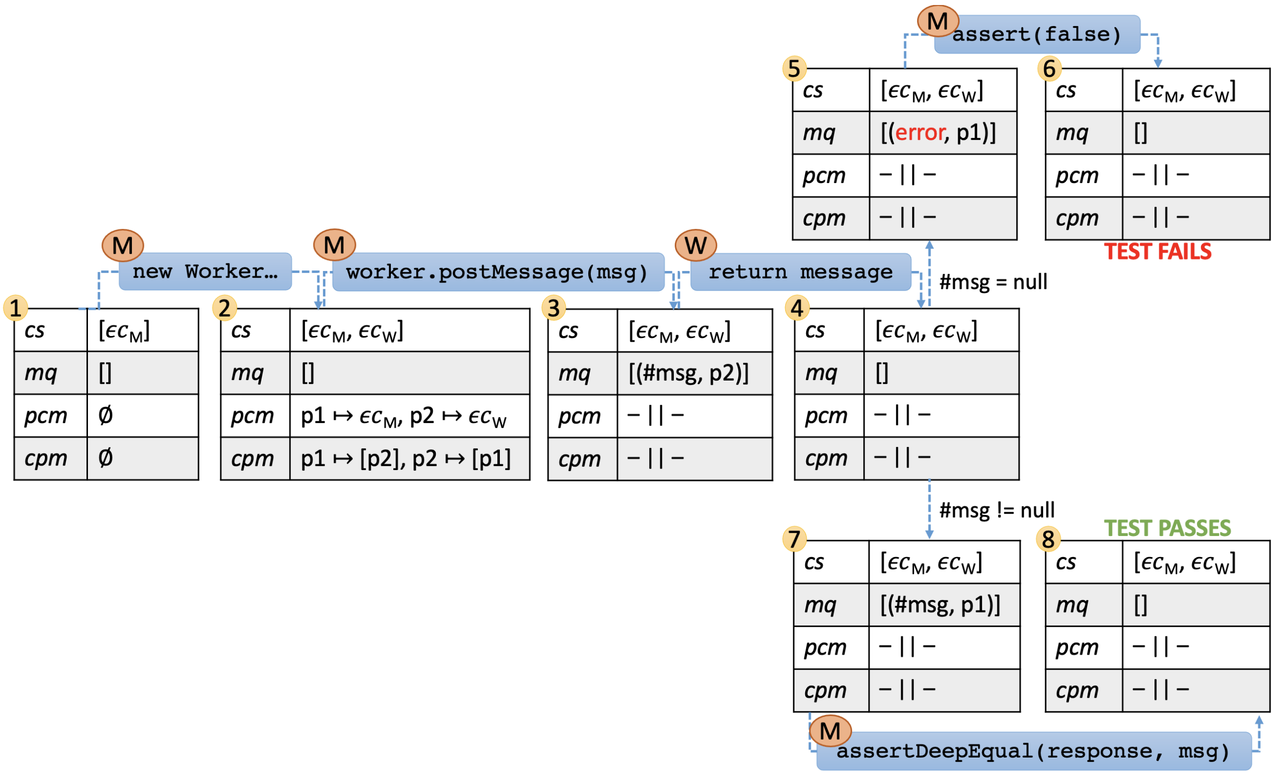

Figure 5.6.: MP-configurations for motivating example (cf. §5.1)

5.5. Concrete MP-semantics

An MP-configuration $mc = \langle cs, mq, pcm, cpm, \ell \rangle$ contains on a configuration sequence cs, a message queue mq, a port-configurations map pcm, a connected-ports map cpm, and a lead configuration ℓ. In Figure 5.7, we provide a diagram to illustrate MP-transitions. We define the MP-semantics in a small-step fashion, with the semantic transition $mc \rightsquigarrow_{\mathsf{MP}} mc'$ representing a single computation step. The semantic transition $mc \rightsquigarrow_{\mathsf{MP}} mc'$ is defined with the help of an auxiliary transition of the form $rc \rightsquigarrow_{\mathsf{MP}} rc'$, referred to as reduced-configuration transition. The idea is that the reduced-configuration transition acts on a single configuration instead of the whole configuration sequence, and generates an action ca denoting an operation on configuration sequences delegated to the general MP-transition.

The MP-semantics first checks whether the current MP-configuration is executing a command inside an atomic block (step 1). If no atomic block is executing, meaning that there is no lead configuration, then the scheduler is invoked to determine which message or configuration is to be processed next (step 2). If the scheduler chooses a message from the message queue, the message is processed (step 3) and the MP-semantics computes a new MP-configuration mc' (step 6). Otherwise, the scheduler chooses a configuration from the configuration sequence and the MP-semantics makes a step on that configuration by applying a reduced-configuration transition (step 4). If an atomic is executing, the reduced-configuration transition is directly applied on the lead configuration with no intervention of the scheduler. Next, the MP-semantics updates the MP-configuration mc based on the obtained E-configuration and configuration action, generating a new configuration sequence (step 5). This may involve adding or removing an E-configuration from the original configuration sequence. The configuration sequence is then updated accordingly and the MP-semantics computes a new MP-configuration

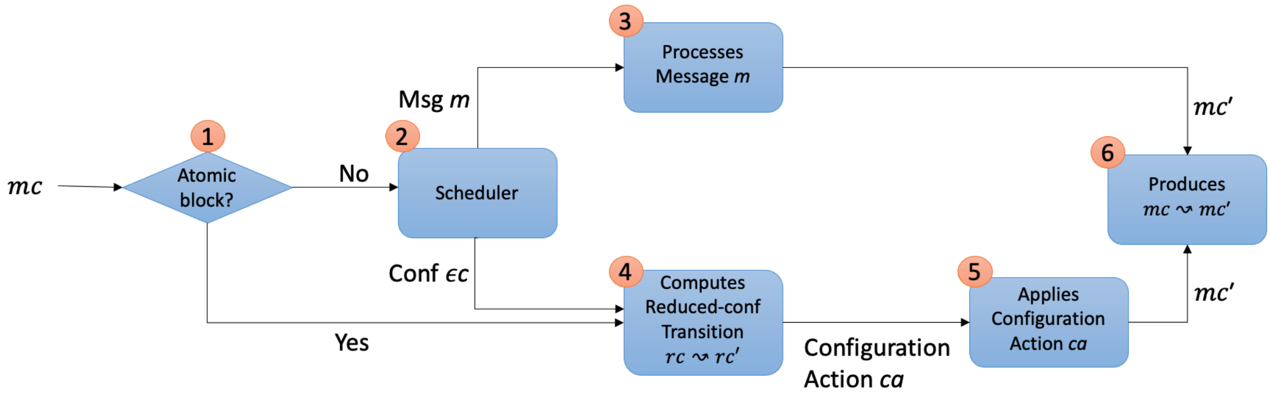

Figure 5.7.: MP-transitions: high-level diagram

mc' (step 6).

In order to update E-configurations, the MP-semantics assumes that an interface is provided by the E-semantics. The concrete rules of the MP-semantics also rely on auxiliary functions to handle mainly port-related operations, such as connecting or disconnecting ports. In the following, we we introduce the E-semantics interface (§5.5.1) and the auxiliary functions of the MP-semantics (§5.5.2). Next, we give the rules of the concrete MP-semantics (§5.5.3) and an example of scheduler (§5.5.4) that could be used to instantiate our MP-semantics. Finally, we define the rules of the reduced MP-semantics (§5.5.5).

5.5.1. Concrete E-semantics Interface

Each MP-configuration contains a sequence of E-configurations, which are opaque to the MP-semantics and are supposed to have their own event loops. Implicitly, we assume that each E-configuration consists of a *language configuration*; an *event-handlers* map, for capturing associations between events and handlers; and an *event queue*, keeping track of events to be processed. For instance, an E-configuration of the E-semantics, $\langle lc, h, q \rangle$, contains a language configuration lc (which corresponds to a JSIL configuration in JaVerT.Click), a handler register h and a continuation queue q. In order to compute sequences of E-configurations, the MP-semantics assumes an interface of E-semantics with three functions: newConf, setVar and final.

1. newConf(vs): creates a new configuration based on the arguments vs that could be defined, for instance, as a tuple $\langle lc, h, q \rangle$, as defined by our E-semantics. The MP-semantics uses this function to set up the memory for a new worker thread. The list of values vs contains, for instance, the path to the worker script that will run in parallel with the current script.

2. setVar$(\epsilon c, x, v)$: updates the value of the variable x to v in the configuration ϵc. This operation is not necessarily performed by the E-semantics, but could actually be the role of the L-semantics, as it consists of a store update. In that case, the E-semantics would just call the respective function provided by the L-semantics. The MP-semantics uses this function, for instance, to update a program variable to a newly generated configuration identifier.

3. final(ϵc): checks wether the event configuration ϵc is final. Intuitively, a configuration is final if there is nothing else to execute at the underlying language configuration. The scheduler of the

MP-semantics uses this function to choose which configuration will make a step.

From now on, we use the prefix ES. to denote a call to a function provided by the E-semantics interface.

5.5.2. Auxiliary Functions of the Concrete MP-semantics

To improve the readability and modularity of our MP-semantics, we define auxiliary functions for updating MP-configurations; their corresponding formal definitions are given in Figure 5.8. We write $+\!\!+$ to denote list concatenation; $cs \setminus \alpha$ to denote the configuration sequence obtained by removing the configuration whose identifier is α from cs; $mq \setminus ps$ to denote the message queue obtained from mq by removing all ports of ps; and $pcm \setminus ps$ and $cpm \setminus ps$ to denote, respectively, the port-configuration map and the connected ports map when removing the ports of ps.

Final: $\mathsf{final}(cs)$ returns true if all the event configurations in cs are final and false otherwise. To know whether a configuration is final, we make use of the underlying $\mathsf{final}(\epsilon c)$ function provided by the E-semantics;

Delete ports: $\texttt{del\_ports}(ps, mq, pcm, cpm)$ deletes the ports of ps from mq, pcm and cpm;

Connect ports: $\texttt{connect\_ports}(p_1, p_2, cpm)$ connects ports p_1 and p_2 in cpm;

Disconnect port: $\texttt{disconnect\_port}(p, cpm)$ disconnects port p in cpm;

Transfer: $\texttt{transfer}(\alpha, ps, pcm)$ transfers each port of ps to configuration α in pcm;

Apply Config Action: $\mathsf{applyAction}(cs, \ell, ca)$ updates the configuration queue cs and the lead configuration ℓ based on the configuration action ca. For instance, if the action is $\texttt{Add}\langle \epsilon c_\alpha \rangle$, the function adds the newly created configuration at the back of the configuration sequence cs and the lead configuration remans unchanged. In contrast, if the action is $\texttt{Hold}\langle \alpha \rangle$, the configuration sequence remains unchanged and the lead configuration becomes the one with identifier α.

5.5.3. Concrete MP-semantics Rules

We define the main rules of the MP-semantics parametrically on the underlying E-semantics and on a *scheduler*. At each semantic step, the scheduler chooses either to make a step using a configuration from the configuration sequence or to process a message from the message queue. This parametricity makes it possible to configure the MP-semantics with a specialised scheduling strategy. For instance, one could choose to use a scheduler that simulates the behaviour of a specific browser, or, alternatively, to use a scheduler that follows a highly interleaving strategy in order to detect atomicity problems. While our semantics does support multiple scheduling strategies, it is not our goal to explore such strategies. Here, we present the general mechanism, leaving the study of effective scheduling strategies and heuristics [21, 80] to future work. We discuss scheduling strategies in more detail shortly.

The MP-semantics has two sources of non-determinism: the E-semantics and the scheduler. The E-semantics may be non-deterministic in that event handlers could be processed in an arbitrary order. The scheduler may be non-deterministic in that it is free to choose either a configuration or a message

Final Configurations

$$\mathsf{final}(cs) \triangleq \begin{cases} \mathsf{true}, & \text{if } cs \text{ is empty} \\ \mathsf{ES.final}(c) \wedge \mathsf{final}(cs'), & \text{if } cs = c : cs' \end{cases}$$

Delete Ports

$$\texttt{del\_ports}(ps, mq, pcm, cpm) \triangleq (mq', pcm', cpm'), \text{ where} \begin{cases} mq' = mq \setminus ps \\ pcm' = pcm \setminus ps \\ cpm = cpm \setminus ps \end{cases}$$

Connect Ports

$$\texttt{connect\_ports}(p_1, p_2, cpm) \triangleq cpm', \text{ where} \begin{cases} ps_1 = cpm(p_1) \\ ps_2 = cpm(p_2) \\ cpm' = cpm[p_1 \mapsto ps_1 + [p_2], p_2 \mapsto ps_2 + [p_1]] \end{cases}$$

Disconnect Port

$$\texttt{disconnect\_port}(p, cpm) \triangleq cpm', \text{ where } cpm' = cpm \setminus p$$

Transfer Ports

$$\texttt{transfer}(\alpha, ps, pcm) \triangleq pcm', \text{ where} \begin{cases} ps = [p_i \mid_{i=0}^{n}] \\ pcm' = pcm[p_0 \mapsto \alpha, ..., p_n \mapsto \alpha] \end{cases}$$

Apply Config Action

$$\mathsf{applyAction}(cs, \ell, ca) \triangleq \begin{cases} (cs, \ell), & \text{if } ca \text{ is } \cdot \\ (cs + [\epsilon c_\alpha], \ell), & \text{if } ca \text{ is } \texttt{Add}\langle \epsilon c_\alpha \rangle \\ (cs \setminus \alpha, \ell), & \text{if } ca \text{ is } \texttt{Rem}\langle \alpha \rangle \\ (cs, \texttt{Conf}\langle \alpha \rangle), & \text{if } ca \text{ is } \texttt{Hold}\langle \alpha \rangle \\ (cs, \cdot), & \text{if} \begin{cases} ca \text{ is } \mathsf{Free}\langle \alpha \rangle \\ \ell \text{ is } \texttt{Conf}\langle \alpha \rangle \end{cases} \\ (cs', \ell), & \text{if} \begin{cases} ca \text{ is } \mathsf{Notify}\langle v, vs \rangle \\ cs = [\epsilon c_i \mid_{i=0}^{n}] \\ \epsilon c_i \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle v, vs \rangle} \epsilon c_i' \mid_{i=0}^{n} \\ cs' = [\epsilon c_i' \mid_{i=0}^{n}] \end{cases} \end{cases}$$

Figure 5.8.: Concrete MP-semantics: Auxiliary Functions

Run Conf - Non Atomic

$$\frac{\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, \epsilon c, cs_{post} \rangle \quad \langle \epsilon c, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca \rangle \quad cs', \ell' = \mathsf{applyAction}(cs_{pre} + [\epsilon c'] + cs_{post}, \cdot, ca)}{\langle cs, mq, pcm, cpm, \cdot \rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm', \ell' \rangle}$$

Run Conf - Atomic

$$\frac{cs_{pre} + [\epsilon c_\alpha] + cs_{post} = cs \quad \ell = \texttt{Conf}\langle \alpha \rangle \quad \langle \epsilon c, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca \rangle \quad cs', \ell' = \mathsf{applyAction}(cs_{pre} + [\epsilon c'] + cs_{post}, \ell, ca)}{\langle cs, mq, pcm, cpm, \ell \rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm', \ell' \rangle}$$

Process Message

$$\frac{\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Msg}\langle ((vs, ps), p), mq' \rangle \quad \alpha = pcm(p) \quad pcm' = \texttt{transfer}(\alpha, ps, pcm) \quad cs_{pre} + [\epsilon c_\alpha] + cs_{post} = cs \quad \epsilon c_\alpha \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \text{ProcessMessage}, vs \rangle} \epsilon c_\alpha' \quad cs' = cs_{pre} + [\epsilon c_\alpha'] + cs_{post}}{\langle cs, mq, pcm, cpm, \cdot \rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm, \cdot \rangle}$$

Figure 5.9.: Message-passing Semantics: $\langle cs, mq, pcm, cpm, \ell \rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm', \ell' \rangle$

to processed. Furthermore, it can choose any of the currently executing configurations, except for the case of atomic blocks for which the currently executing configuration must always be chosen. The rules of the MP-semantics are given in Figure 5.9 and are explained in the following.

[Run Configuration - Non Atomic Block] This rule is applied when there is no leading configuration, meaning that the semantics is not executing an atomic block. The scheduler chooses a configuration to run by returning $\mathsf{Conf}\langle cs_{pre}, \epsilon c, cs_{post}\rangle$. This indicates that the MP-semantics must take an E-semantics step on the configuration ϵc rather than process a pending message. The scheduler also gives the lists of configurations coming before and after ϵc, cs_{pre} and cs_{post} respectively. The MP-semantics then applies the reduced-configuration transition to the chosen configuration ϵc. Finally, the resulting configuration $\epsilon c'$ replaces ϵc in the configuration sequence and the configuration action ca is applied to the obtained sequence using the semantic function applyAction.

[Run Configuration - Atomic Block] In contrast to the previous rule, this rule does not make use of the scheduler, as it is applied when the MP-semantics is executing an atomic block. In this case, the MP-semantics starts by obtaining the leading configuration ϵc_α. Then, it applies the reduced-configuration transition to ϵc_α proceeding analogously to the previous rule.

[Process Message] The scheduler can also choose to process a message from the message queue by returning $\mathsf{Msg}\langle((vs, ps), p), mq'\rangle$. Intuitively, the processing of a message in the MP-semantics means that the message is forwarded to the receiver configuration ϵc. Then, the event loop of ϵc takes care of executing the appropriate handlers. The tuple (vs, ps) is the message to be sent, p is the target port, and mq' is the resulting message queue. The MP-semantics then: **(1)** obtains the configuration ϵc to which the message is addressed; **(2)** transfers the ports of ps to that configuration using the auxiliary function $\texttt{transfer}(\alpha, ps, pcm)$; **(3)** triggers the ProcessMessage event with the arguments vs supplied in the message on the target configuration ϵc; and **(4)** updates the configuration sequence accordingly.

5.5.4. Scheduler

The MP-semantics is parametric on a scheduler that can choose either a configuration to run or a message to be processed. Our semantics supports both deterministic and non-deterministic schedulers, branching on all possible scheduling results. In Figure 5.10 we show an example of a scheduler that prioritises configuration steps over the processing of messages. In particular, the scheduler always chooses the first non-final configuration if such a configuration exists, meaning that it chooses the first configuration that still has code to be executed at the underlying language level. If all configurations are final, the scheduler re-arranges the message queue so that port-transferring messages appear first and then chooses the first message of the re-arranged queue to be processed.

In a nutshell, the scheduler first chooses configuration steps, then port-transferring messages, and finally non-port-transferring messages. Most browsers seem to follow a similar strategy to this scheduler [67]. One could, however, instantiate the MP-semantics with other scheduling strategies and potentially find unexpected behaviours. We explain the rules below.

[Configuration Scheduled] The scheduler chooses a configuration if there is a way of splitting the

$$\frac{cs_{pre} \mathbin{+\!\!+} [\epsilon c] \mathbin{+\!\!+} cs_{post} = cs \quad \mathsf{final}(cs_{pre}) \quad !\mathsf{ES.final}(\epsilon c)}{\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, \epsilon c, cs_{pos}\rangle}\ \textsc{Configuration Scheduled}$$

$$\frac{mq' = mq \triangleright (\lambda(vs, ps) \cdot ps \neq []) \mathbin{+\!\!+} mq \triangleright (\lambda(vs, ps) \cdot ps = []) \quad m :: mq'' = mq' \quad \mathsf{final}(cs)}{\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Msg}\langle m, mq''\rangle}\ \textsc{Message Scheduled}$$

Figure 5.10.: MP-semantics: Scheduler Example

configuration queue cs into three parts: a list of configurations cs_{pre} of final configurations, a non-final configuration ϵc and a remaining list of configurations cs_{post}.

[Message Scheduled] The scheduler chooses a message only if all configurations in cs are final (given by $\mathsf{final}(cs)$). Note that this makes the scheduler deterministic. However, if we were to remove the $\mathsf{final}(cs)$ condition from the premise, we would obtain a non-deterministic scheduler. The scheduler rearranges the message queue mq so that messages with transferred ports are processed before the ones with no transferred ports. We use $mq \triangleright f$ to denote all the elements of mq that satisfy the predicate f. Finally, the scheduler returns a pair with the first message m of the resulting queue and the queue with the pending messages mq''.

Note that the given scheduler does not handle non-termination. Supposing that the scheduler chooses a configuration to make a step and that step leads to an infinite loop, there would be a lack of global progress as the scheduler would keep choosing that configuration until it is final, meaning that there are no more steps to be done. We choose to run each configuration up to completion to model the concurrency model of JavaScript, which is based on event loop concurrency. One could develop a sophisticated mechanism in future to guarantee that every thread will eventually make forward progress even in the presence of infinite loops.

5.5.5. Reduced MP-semantics Rules

The reduced-configuration transitions are given in Figure 5.11. For readability, we omit the configuration identifier α if it is not used. Furthermore, in the rules, we conflate output reduced configuration tuples with the generated configuration action, writing $\langle \epsilon c, mq, pcm, cpm, ca\rangle$ instead of $(\langle \epsilon c, mq, pcm, cpm\rangle, ca)$. We omit the configuration action ca when it is not present, meaning that it is equal to $\cdot$. The rules of the reduced semantics are explained below; they rely on the E-semantics interface and on the auxiliary semantic functions, defined in §5.5.1 and §5.5.2.

[E-semantics Transition] If the E-semantics generates the message-passing primitive $\cdot$, the reduced semantics simply updates its inner configuration accordingly.

[Post Message] If the E-semantics generates the primitive $\mathsf{send}\langle vs, ps, p_1, p_2\rangle$, the reduced semantics enqueues the message (vs, ps) in the message queue. Messages are added at the back of the message queue. For the send operation to succeed, p_1 must be connected with p_2, meaning that p_2 is in the set of ports connected with p_1, formally: $p_2 \in cpm(p_1)$.

[New Execution] If the E-semantics generates the primitive $\mathsf{create}\langle x, vs\rangle$, the reduced semantics makes use of the auxiliary function $\mathsf{newConf}(vs)$ for creating a fresh configuration. The value

$$\textsc{E-semantics Transition}\quad \frac{\epsilon c \rightsquigarrow^{\cdot}_{\mathsf{E}} \epsilon c'}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm\rangle}$$

$$\textsc{Post Message}\quad \frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{send}\langle vs, ps, p_1, p_2\rangle \qquad p_2 \in cpm(p_1) \quad mq' = mq +\!\!+ [((vs, ps), p_2)]}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm, cpm\rangle}$$

$$\textsc{New Execution}\quad \frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{create}\langle x, vs\rangle \qquad \epsilon c''_{\alpha} = \mathsf{ES.newConf}(vs) \qquad \epsilon c''' = \mathsf{ES.setVar}(\epsilon c', x, \alpha) \quad ca = \mathtt{Add}\langle \epsilon c''_{\alpha}\rangle}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c''', mq, pcm, cpm, ca\rangle}$$

$$\textsc{Terminate Execution}\quad \frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{terminate}\langle \alpha\rangle \quad ps = pcm \triangleright \alpha \qquad (mq', pcm', cpm') = \mathtt{del\_ports}(ps, mq, pcm, cpm) \qquad ca = \mathtt{Rem}\langle \alpha\rangle}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca\rangle}$$

$$\textsc{New Port}\quad \frac{\epsilon c_{\alpha} \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_{\alpha} \quad \mathrm{p} = \mathsf{newPort}\langle\rangle \quad p \text{ is fresh} \qquad pcm' = pcm[p \mapsto \alpha]}{\langle \epsilon c_{\alpha}, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_{\alpha}, mq, pcm', cpm\rangle}$$

$$\textsc{Get Connected Ports}\quad \frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{getConnected}\langle x, p\rangle \qquad ps = cpm(p) \quad \epsilon c'' = \mathsf{ES.setVar}(\epsilon c', x, ps)}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'', mq, pcm, cpm\rangle}$$

$$\textsc{Connect Ports}\quad \frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{connect}\langle p_1, p_2\rangle \qquad cpm' = \mathtt{connect\_ports}(p_1, p_2, cpm)}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm'\rangle}$$

$$\textsc{Disconnect Port}\quad \frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{disconnect}\langle p\rangle \qquad cpm' = \mathtt{disconnect\_port}(p, cpm)}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm'\rangle}$$

$$\textsc{Begin Atomic}\quad \frac{\epsilon c_{\alpha} \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_{\alpha} \quad \mathrm{p} = \mathsf{beginAtomic} \quad ca = \mathtt{Hold}\langle \alpha\rangle}{\langle \epsilon c_{\alpha}, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_{\alpha}, mq, pcm, cpm, ca\rangle}$$

$$\textsc{End Atomic}\quad \frac{\epsilon c_{\alpha} \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_{\alpha} \quad \mathrm{p} = \mathsf{endAtomic} \quad ca = \mathsf{Free}\langle \alpha\rangle}{\langle \epsilon c_{\alpha}, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_{\alpha}, mq, pcm, cpm, ca\rangle}$$

$$\textsc{Notify All}\quad \frac{\epsilon c_{\alpha} \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_{\alpha} \quad \mathrm{p} = \mathsf{notifyAll}\langle v, vs\rangle}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm, \mathsf{Notify}\langle v, vs\rangle\rangle}$$

Figure 5.11.: Reduced Semantics: $\langle \epsilon c, mq, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca\rangle$

list vs contains the parameters required for setting up a new configuration. The MP-semantics also makes use of the auxiliary function $\mathsf{setVar}(\epsilon c, x, \alpha)$ to assign the identifier of the newly created configuration α to the variable x in the currently executing configuration ϵc. This reduced semantic transition generates the configuration action $\mathtt{Add}\langle \epsilon c''_{\alpha}\rangle$ to signal to the MP-semantics that the newly created configuration is to be added to configuration sequence.

[Terminate Execution] If the E-semantics generates the primitive $\mathsf{terminate}\langle \alpha\rangle$, the reduced semantics is responsible for immediately terminating the configuration with identifier α. This requires removing all pending messages addressed to that configuration as well as removing the ports ps owned by that configuration. We use $pcm \triangleright \alpha$ to denote the ports belonging to the configuration with identifier α and the auxiliary function $\mathtt{del\_ports}(ps, mq, pcm, cpm)$ to update the message queue, port-configurations map, and connected-ports map, removing all references to α and its associated ports. Similarly to the previous rule, the sequence of configurations is not updated at this stage. Instead, the transition generates the configuration action $\mathtt{Rem}\langle \alpha\rangle$ to signal to the MP-semantics that the configuration with identifier α is to be removed from the configuration sequence.

[New Port] If the E-semantics generates the primitive $\mathsf{newPort}\langle\rangle$, the reduced semantics creates a fresh port in the current configuration. The reduced semantics then updates the port-configurations map pcm by adding the entry (p, α).

[Connect Ports] If the E-semantics generates the primitive $\mathsf{connect}\langle p_1, p_2 \rangle$, the reduced semantics connects the ports p_1 and p_2 using the auxiliary function $\mathtt{connect\_ports}(p_1, p_2, cpm)$. Note that communication is bi-directional, meaning that messages sent from p_1 are delivered to p_2 and vice-versa.

[Disconnect Port] If the E-semantics generates the primitive $\mathsf{disconnect}\langle p \rangle$, the reduced semantics disconnects p from all the ports to which it is currently connected with the help of the auxiliary function $\mathtt{disconnect\_port}(p, cpm)$.

[Get Connected Ports] If the E-semantics generates the primitive $\mathsf{getConnected}\langle x, p \rangle$, the reduced semantics obtains the ports ps connected with p by accessing the connected-ports map cpm. Finally, the reduced semantics calls $\mathsf{setVar}(c, x, ps)$ to update the value of the variable x of the E-semantics configuration with ps.

[Begin Atomic] If the E-semantics generates the primitive beginAtomic, the reduced semantics must ensure that there is no interleaving of configurations until an endAtomic primitive is found. The reduced semantics simply generates the configuration action $\mathtt{Hold}\langle \alpha \rangle$ that signals to the MP-semantics that the configuration with identifier α is to be turned into a leading configuration.

[End Atomic] Works analogously to the previous rule. The reduced semantics generates the configuration action $\mathsf{Free}\langle \alpha \rangle$, indicating that the scheduler can run normally and the configuration α does not need to be chosen.

[Notify All] If the E-semantics generates the primitive $\mathsf{notifyAll}\langle v, vs \rangle$, the reduced semantics generates the configuration action $\mathsf{Notify}\langle v, vs \rangle$, so that the event v is triggered on all configurations with arguments vs.

5.6. Symbolic Message-passing Semantics

We instantiate the MP-semantics with a symbolic E-semantics to obtain a symbolic MP-semantics. We assume that the underlying symbolic E-semantics operates on symbolic configurations $\widehat{ec} \in \widehat{\mathcal{EC}}$, and that the underlying language configuration of $\widehat{ec}$ enables the use of symbolic values, $\hat{v} \in \widehat{\mathcal{V}}$ and symbolic variables, $\hat{x} \in \hat{\mathcal{X}}$.

Accordingly, MP-configurations are composed of sequences of symbolic event configurations, $\hat{cs} \in \hat{\mathcal{CS}}$. As the messages handled by the MP-semantics are created from the values provided by the underlying E-semantics, these messages, $\hat{m} \in \hat{\mathcal{M}}$, can hold both symbolic and concrete values. Symbolic message queues, $\hat{mq} \in \hat{\mathcal{MQ}}$, can, therefore, be composed of both concrete and symbolic messages. In contrast to messages, port and configuration identifiers are restricted to concrete values. Hence, the port-configurations map $pcm \in \mathcal{PCM}$, connected-ports map $cpm \in \mathcal{CPM}$, and lead configuration $\ell \in \mathcal{L}$ used by the symbolic MP-semantics are exclusively composed of concrete values. In summary, symbolic MP-configurations are of the form $\widehat{mc} = \langle \hat{cs}, \hat{mq}, pcm, cpm, \ell \rangle$.

The rules of the symbolic MP-semantics, like the concrete ones, also rely on an E-semantics interface and on a set of auxiliary functions. In the following, we introduce the symbolic E-semantics interface (§5.6.1) and the auxiliary functions of the symbolic MP-semantics (§5.6.2). Finally, we discuss the rules of the symbolic MP-semantics (§5.6.3) and correctness results (§5.6.4).

5.6.1. Symbolic E-semantics Interface

The symbolic E-semantics interface includes all functions defined for the concrete one: $\mathsf{newConf}$, setVar and final. Additionally, the symbolic MP-semantics assumes that each symbolic event configuration provides a *path condition* π, which is a first-order quantifier-free formula accumulating the constraints on the symbolic inputs that direct the execution along that path. In JaVerT.Click, the path condition is provided by the underlying language configuration. For instance, a JSIL symbolic configuration $\langle \hat{s}, \hat{\mu}, \hat{cs}, i, \pi \rangle$ includes a symbolic store $\hat{s}$, a symbolic memory $\hat{h}$, a symbolic call stack $\hat{cs}$, the index of the current command i, and a path condition π. For the MP-semantics to keep track of the path condition across multiple E-configurations, the E-semantics needs to provide two additional functions: assume and $\mathsf{pc}()$. In our implementation of the E-semantics, both the assume and $\mathsf{pc}()$ simply call the respective functions provided by the L-semantics interface (see §4.6.1).

1. $\mathsf{assume}(\widehat{\epsilon c}, \pi) = \widehat{\epsilon c}'$, where $\widehat{\epsilon c}'$ is obtained from $\widehat{\epsilon c}$ by extending its path condition with the formula π, if such an extension is satisfiable.

2. $\mathsf{pc}(\widehat{\epsilon c}) = \pi$, where π is the path condition computed in the current branch of configuration $\widehat{\epsilon c}$.

5.6.2. Auxiliary Functions of the Symbolic MP-semantics

The symbolic MP-semantics relies on exactly the same auxiliary functions of the concrete MP-semantics: final, `del_ports`, `connect_ports`, `disconnect_port`, `transfer` and $\mathsf{applyAction}$ (c.f. §5.5.2). However, the function $\mathsf{applyAction}$ needs to be adapted for symbolic execution in order to support the $\mathsf{Assume}\langle \mathtt{f} \rangle$ configuration action. More specifically, given a symbolic configuration sequence $\hat{cs}$ and a lead configuration ℓ, the $\mathsf{applyAction}$ function propagates the assumption to all configurations by using the function $\mathsf{assume}(\widehat{\epsilon c}, \mathtt{f})$ that must be provided by the E-semantics. The lead configuration remains unchanged, as the configuration action $\mathsf{Assume}\langle \mathtt{f} \rangle$ does not affect atomic blocks.

$$\mathsf{applyAction}(\hat{cs}, \ell, \mathsf{Assume}\langle \mathtt{f} \rangle) = ([\mathsf{ES.assume}(\widehat{\epsilon c}_i, \mathtt{f}) \mid_{i=0}^{n}], \ell), \text{where } \hat{cs} = [\widehat{\epsilon c}_i \mid_{i=0}^{n}]$$

Note that the path condition of each E-configuration is equal to the others, as all path conditions are updated together every time a constraint is generated from any existing E-configuration. One could, alternatively, choose to have a single path condition which would be provided by the symbolic MP-semantics. We prefer our approach because we believe that it is not the role of the MP-semantics to manage path conditions; a path condition should be part of the underlying language configuration. This way, the underlying language semantics is independent of both the E-semantics and the MP-semantics in terms of the symbolic analysis. This allows us to use JaVerT.Click with or without the E-semantics and MP-semantics, depending if we need to use events or message-passing-related features.

5.6.3. Symbolic MP-semantics Rules

Besides the rules defined for the concrete MP-semantics, the symbolic MP-semantics provides an additional rule for maintaining the same path condition in all running configurations. In particular, every time a new constraint `f` is added to the path condition of one of the configurations in the configuration sequence , the MP-semantics is notified so that it can update the other configurations of the configuration sequence. This is important to avoid inconsistencies between different configurations. For instance, if we assume that a symbolic variable $\hat{x}$ is of type `object` in the main thread, $\hat{x}$ should also have type `object` in the worker threads. In Figure 5.12, we give the Assume rule defined at the level of the reduced semantics.

$$\textsc{Assume}\quad \frac{\widehat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\hat{\mathrm{p}}} \widehat{\epsilon c}' \quad \hat{\mathrm{p}} = \mathsf{assume}\langle\mathtt{f}\rangle \quad ca = \mathsf{Assume}\langle\mathtt{f}\rangle}{\langle \widehat{\epsilon c}, \hat{mq}, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \widehat{\epsilon c}', \hat{mq}, pcm, cpm, ca\rangle}$$

Figure 5.12.: Assume Rule of the Symbolic reduced semantics

This rule works analogously to the rules Begin Atomic and End Atomic (§5.5.5). If the E-semantics generates the primitive $\mathsf{assume}\langle\mathtt{f}\rangle$, meaning that the formula `f` holds for the currently executing event configuration, the reduced semantics computes a configuration action $\mathsf{Assume}\langle\mathtt{f}\rangle$ that is then processed at the MP-semantics level. The MP-semantics processes the action $\mathsf{Assume}\langle\mathtt{f}\rangle$ by adding the formula `f` to the path condition of each configuration in the configuration sequence $\hat{cs}$, as explained in §5.6.2.

5.6.4. Correctness

We establish the correctness of a symbolic MP-semantics w.r.t. a concrete MP-semantics using analogous notions of Directed Soundness and Directed Completeness to the ones defined for our E-semantics (c.f. §4.6.4): Directed Soundness holds when every symbolic trace over-approximates all concrete traces that follow its execution path, while Directed Completeness holds if every symbolic trace has at least one valid concretisation. The former property guarantees the absence of false-positive bug-reports, meaning that if a bug is reported symbolically, it must also be reported concretely. For the MP-semantics to be correct it must satisfy both properties [33, 35, 107, 32, 91]. Because the E-semantics is opaque to the MP-semantics, we assume that the E-semantics satisfies directed soundness and directed completeness. In §4.6.4, we proved this result for the E-semantics of JaVerT.Click.

The MP-semantics, like the E-semantics, also relies on *logical environments* of the form $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$ to relate symbolic and concrete MP-configurations. We extend the interpretation function $\mathcal{I}_\varepsilon$ (c.f. §4.6.4) to all message-passing structures defined in §5.3. The respective definitions are shown in Figure 5.13. For example, $\mathcal{I}_\varepsilon(\langle \hat{cs}, \hat{mq}, pcm, cpm, \ell\rangle) \triangleq \langle \mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm, \ell\rangle$. We assume that interpretation is preserved by the functions of the E-semantics interface; for example, that $\mathsf{final}(\widehat{\epsilon c}) \Leftrightarrow \mathsf{final}(\mathcal{I}_\varepsilon(\widehat{\epsilon c}))$.

We also extend the *concretisation function* $\mathcal{M}_\pi()$ defined for L-configurations and E-configurations to MP-configurations. Hence, given a symbolic MP-configuration $\widehat{mc}$, $\mathcal{M}_\pi(\widehat{mc})$ denotes the set of concrete MP-configurations obtained via interpretations of $\widehat{mc}$ that satisfy the path condition π. More formally, $mc \in \mathcal{M}_\pi(\widehat{mc})$ if there is a *logical environment* ε that evaluates $\widehat{mc}$ to mc and π to *true*.

$\mathcal{CS}$ - Empty
$\mathcal{I}_\varepsilon(\emptyset) \triangleq \emptyset$

$\mathcal{CS}$ - Composition
$\mathcal{I}_\varepsilon(\hat{cs}_1 \uplus \hat{cs}_2) \triangleq \mathcal{I}_\varepsilon(\hat{cs}_1) \uplus \mathcal{I}_\varepsilon(\hat{cs}_2)$

$\mathcal{CS}$ - Cell
$\mathcal{I}_\varepsilon([(\hat{\epsilon c}, \alpha)]) \triangleq [(\mathcal{I}_\varepsilon(\hat{\epsilon c}), \alpha)]$

$\mathcal{MQ}$ - Empty
$\mathcal{I}_\varepsilon([]) \triangleq []$

$\mathcal{MQ}$ - Non-Empty
$\mathcal{I}_\varepsilon(([\hat{v}_1, ..., \hat{v}_n], ps), p) \triangleq (([\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n), ps), p)]$

MP Confs
$\mathcal{I}_\varepsilon(\langle \hat{cs}, \hat{mq}, pcm, cpm, \ell \rangle) \triangleq \langle \mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm, \ell \rangle$

MP primitive - Send
$\mathcal{I}_\varepsilon(\mathsf{send}\langle \hat{vs}, ps, p_1, p_2 \rangle) \triangleq \mathsf{send}\langle \mathcal{I}_\varepsilon(\hat{vs}), ps, p_1, p_2 \rangle$

MP primitive - Create
$\mathcal{I}_\varepsilon(\mathsf{create}\langle \hat{x}, \hat{vs} \rangle) \triangleq \mathsf{create}\langle \mathcal{I}_\varepsilon(\hat{x}), \mathcal{I}_\varepsilon(\hat{vs}) \rangle$

Configuration Actions (Add)
$\mathcal{I}_\varepsilon(\mathtt{Add}\langle \hat{\epsilon c}, \alpha \rangle) \triangleq \mathtt{Add}\langle \mathcal{I}_\varepsilon(\hat{\epsilon c}), \alpha \rangle$

Configuration Actions (Notify)
$\mathcal{I}_\varepsilon(\mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle) \triangleq \mathsf{Notify}\langle \mathcal{I}_\varepsilon(\hat{v}), \mathcal{I}_\varepsilon(\hat{vs}) \rangle$

Figure 5.13.: Interpretation of MP-semantics Structures

Definition 5.1 formalises the correctness of the underlying E-semantics. In §4.6.4, we proved that this holds for the E-semantics of JaVerT.Click (Theorem 4.1).

Definition 5.1 (Correctness Criteria - Symbolic E-semantics)**.**

E-Directed-Soundness
$$\widehat{\epsilon c} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathsf{E}} \widehat{\epsilon c}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')) \wedge (\varepsilon, c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \wedge c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} c' \implies (\varepsilon, c') \in \mathcal{M}_\pi(\widehat{\epsilon c}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$$

E-Directed-Completeness
$$\widehat{\epsilon c} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathsf{E}} \widehat{\epsilon c}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')) \wedge (\varepsilon, c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \implies \exists \, \mathrm{p}, c'. \; c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} c'$$

Theorem 5.1 formalises the two properties for the MP-semantics. We use $\widehat{mc}$ and mc to denote symbolic and concrete MP-configurations. Directed Soundness states that, given symbolic and concrete MP-transitions, respectively $\widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}'$ and $mc \rightsquigarrow_{\mathsf{MP}} mc'$, if we know that **(1)** the current path condition satisfies the final path condition of $\widehat{mc}'$ (given by $\pi \Rightarrow \mathsf{pc}(\widehat{mc}')$) and **(2)** the initial concrete MP-configuration mc is in the models of $\widehat{mc}$ filtered by π (given by $(\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc})$), we can guarantee that the final concrete MP-configuration mc' is in the *models* of the final symbolic MP-configuration $\widehat{mc}'$ under path condition π.

Directed Completeness is formalised analogously, but guarantees that there is at least one concrete MP-transition given that the initial concrete MP-configuration mc is in the set of models of the initial symbolic MP-configuration $\widehat{mc}$.

Theorem 5.1 (Correctness of the Symbolic MP-semantics)**.**

MP-Directed-Soundness
$$\widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}' \wedge \pi \Rightarrow \mathsf{pc}(\widehat{mc}') \wedge (\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc}) \wedge mc \rightsquigarrow_{\mathsf{MP}} mc' \implies (\varepsilon, mc') \in \mathcal{M}_\pi(\widehat{mc}')$$

MP-Directed-Completeness
$$\widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}' \wedge \pi \Rightarrow \mathsf{pc}(\widehat{mc}') \wedge (\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc}) \implies \exists \, mc'. \; mc \rightsquigarrow_{\mathsf{MP}} mc'$$

The proof of Theorem 5.1 is done by case analysis on the rules for the MP-semantics. We assume that Directed Soundness and Directed Completeness hold for the E-semantics and prove these two properties for the MP-semantics. All definitions and details of the proof of Theorem 5.1 can be found in the appendix (§B).

6. Reference Implementations of Web APIs

The main motivation of this work is to perform symbolic analysis on programs calling APIs that are either event-based or rely on a concurrent message-passing model. To achieve our goal, we target 5 Web APIs: `DOM` [136], JS Promises [28], JS async/await [29], WebMessaging [140] and WebWorkers [133]. Our reference implementations are trustworthy in the sense that they are thoroughly tested against their respective test suites and they follow their respective standards line-by-line with the exception of the JS async/await API.

These APIs have become popular inside the JavaScript community and most bugs in client-side applications are related to their usage [94]. We introduced an E-semantics (Chapter 4) and an MP-semantics (Chapter 5) to capture the essence of Web APIs relying on events or a message-passing model. The listed APIs are implemented directly in JavaScript making use of the primitive operations provided by the E-semantics and the MP-semantics. For each API, we identify the minimal set of primitives on which it relies.

Outline. We first introduce our reference implementation of the DOM, which covers the DOM Core Level 1 (§6.1) and DOM Events (§6.2). Next, we introduce our reference implementations of two APIs from the ECMAScript standard: JS Promises (§6.3) and JS async/await (§6.4). Finally, we present our implementation of a fragment of the HTML standard covering the WebMessaging (§6.5) and WebWorkers (§6.6) APIs. For each API, we give an overview by introducing its main interfaces; then, we detail the structure of our reference implementation focusing on their usage of the E-semantics and MP-semantics.

6.1. DOM Core Level 1

The DOM Core Level 1 API [130] is the first version of the DOM API. It describes how XML/HTML documents are internally represented as DOM trees and defines a range of methods for manipulating these trees. DOM trees comprise several different types of DOM nodes and are subject to a number of topological constraints restricting the ways in which these nodes can form a valid DOM tree. For instance, the root node of every DOM tree must have type `Document` and can have at most one child of type `Element`. Elements, on the other hand, can have multiple child nodes of different types, such as `Text` and `Element`.

6.1.1. API Overview

The DOM standard defines interfaces describing the structure of every type of DOM node in an object-oriented style. For every node type, the standard specifies the fields and methods exposed by the nodes of that type. Furthermore, as in standard OO languages, each node type might *inherit* from another node type. The `Node` interface is the parent of all other interfaces, such as `Element`, `ProcessingInstruction`, `Comment`, `EntityReference`, `Attr` and `Text`.

In Figure 6.1, we show the IDL specification of the `Element` interface taken from the standard. Every `Element` node is also a `Node`, meaning that it exposes all fields and methods defined in the `Node` interface. Additionally, every `Element` object exposes:

- the field `tagName`, which represents the name of the element;
- the methods `getAttribute`, `setAttribute` and `removeAttribute`, for retrieving, adding/modifying and removing an attribute by its name;
- the methods `getAttributeNode`, `setAttributeNode` and `removeAttributeNode`, which are analogous to the methods listed above but take the `Attr` object instead of its name;
- the method `getElementsByTagName` for obtaining a `NodeList` containing all descendant elements with a given tag name;
- the method `normalize`, which converts the subtree of this element into a normal form, where none of the `Text` nodes is empty and there are no adjacent `Text` nodes.

```
interface Element : Node {
  readonly attribute DOMString tagName;
  DOMString getAttribute(DOMString name);
  void setAttribute(DOMString name, DOMString value) raises (DOMException);
  void removeAttribute(DOMString name) raises (DOMException);
  Attr getAttributeNode(DOMString name);
  Attr setAttributeNode(Attr newAttr) raises (DOMException);
  Attr removeAttributeNode(Attr oldAttr) raises (DOMException);
  NodeList getElementsByTagName(DOMString name);
  void normalize();
};
```

Figure 6.1.: IDL specification of the `Element` interface

6.1.2. API Reference Implementation

Our reference implementation of the DOM Core Level 1 covers 100% of the respective standard, meaning that we implement all exposed interfaces, including the `Node` interface and its subtypes. We implement the DOM Core Level 1 in JavaScript (ES5 Strict), encoding DOM objects as JS objects. In particular, each type of DOM node is mapped to the JS constructor function in charge of creating the nodes of that type. Also, we emulate class-based inheritance, which is used to describe DOM nodes in the standard, using the prototype inheritance of JS, by storing the methods shared by all nodes of a given type in their (shared) prototype.

In Figure 6.2, we show a fragment of the object graph from our JavaScript implementation considering the `Element` interface. Besides exposing the property `tagName`, all `Element` objects directly define the properties corresponding to the fields of the `Node` interface (e.g. `nodeName`, `ownerDocument`, etc). The methods of the `Element` interface are stored in the object `ElemProto`, the prototype of all `Element` objects, and the `Node` methods are stored in `NodeProto`, which is the prototype of `ElemProto`.

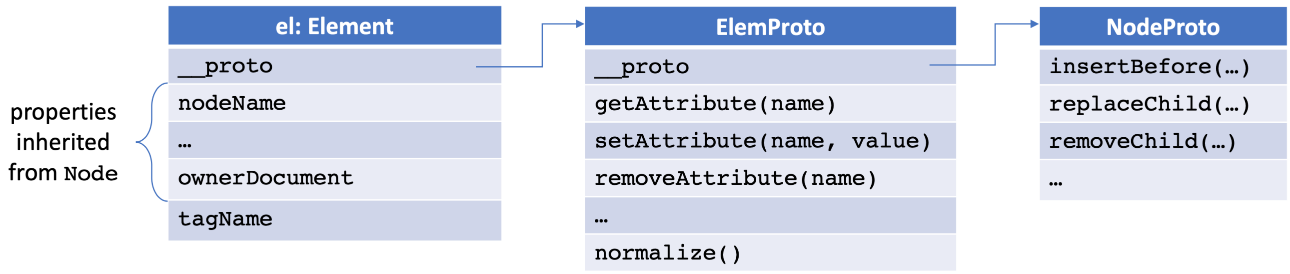

Figure 6.2.: JavaScript object graph for the `Element` interface

In the following, we describe two technical challenges related to the DOM Core Level 1 reference implementation: handling live collections and dynamic parsing of XML/HTML documents. Finally, we show the line-by-line correspondence of our implementation and the DOM Core Level 1 standard.

Live Collections. The `NodeList` interface describes the so-called DOM *live collections*. A live collection is a special data structure defined in the DOM API that automatically reflects changes that occur in its associated document. For instance, the `getElementsByTagName` method from the above-mentioned `Element` interface returns a live collection containing the DOM nodes that match the supplied tag name. Working with live collections is error-prone and requires particular attention. Consider, for example, the following program:

```
var divs = body.getElementsByTagName("div");
for (var i = 0; i < divs.length; i++)
  { body.appendChild(document.createElement("div")) }
```

This program iterates over the initial collection of `div` nodes in the DOM tree rooted at `body`. On each iteration, it creates a new `div` node and inserts it into the original tree. However, this new `div` is also inserted into the live collection `divs`, whose length automatically increases by one, causing the program to loop forever.

The `NodeList` interface defines the field `length`, for obtaining the length of a node list, and the method `item(i)` for accessing its i-th element. In JavaScript, we implement node lists *lazily* in that we recompute the contents of a given node list every time it is inspected. This we achieve by extending `NodeList` objects with an internal `compute` function, used to compute its contents. We call `compute` at every invocation of the `item` method, and associate the `length` property of every node list with a JavaScript *getter* that also calls `compute` before checking the the length of the corresponding node list. As an optimisation, we cache computed live collections by associating each node list with a unique identifier and maintaining a global array of computed node lists. However, whenever there is any update to the DOM tree, all cached live collections are invalidated and will be re-computed the next time they are inspected.

Line-by-Line Closeness. The DOM Core Level 1 standard, unlike the other standards supported by JaVerT.Click, is written in a more descriptive style, meaning that the functions exposed by the DOM Core Level 1 are not described in pseudocode style. Hence, we cannot establish a line-by-line correspondence between the DOM Core Level 1 standard and our reference implementation. However, we are confident that our reference implementation of the DOM Core Level 1 API follows precisely the standard because we pass all available tests from the official test suite, as we will detail in Chapter 7. In contrast to the DOM Core Level 1 standard, functions exposed by the ECMAScript, HTML5 and DOM Living standards are defined step-by-step in pseudocode style, allowing us to establish a line-by-line correspondence.

6.2. DOM UI Events

The DOM Events API [136] describes the DOM event model. In particular, it provides the mechanism for programmers to register event listeners, and explains how these listeners are collected and executed every time a DOM event gets triggered either by the environment (for example, via user events and browser events) or programatically.

6.2.1. API Overview

The DOM Events API is composed of three main interfaces: `Event`, denoting any DOM event; `EventTarget`, denoting a target (e.g.: a DOM node) to which an event can be dispatched when something has occurred; and `CustomEvent`, denoting an event with additional data, such as the current time. There are further interfaces to denote types of events, such as `MouseEvent`, `KeyboardEvent` and `FocusEvent`.

Like the DOM Core Level 1 standard, the structure of the DOM Events standard is described in object-oriented style with each interface exposing a set of fields and methods. In Figure 6.3, we show the IDL specification of the `EventTarget` interface as defined by the standard which includes:

- a `constructor` for creating an object of type `EventTarget`;
- the methods `addEventListener` and `removeEventListener`, for registering and deregistering a given listener to an event;
- the method `dispatchEvent` for synchronously dispatching an event. Returns `false` if either the event's `cancelable` flag is active or if any of the handlers invoked called the `preventDefault`() method exposed by the `Event` interface. Otherwise, returns `true`.

DOM Event Dispatch. At the core of the UI Events API is the DOM `Dispatch` algorithm, which precisely describes the process of collecting and executing event listeners every time a DOM event gets triggered. The DOM Living standard includes the pseudo-code of the `Dispatch` algorithm, detailing all the steps that are performed when dispatching a DOM event, for instance, by calling the `dispatchEvent` function. It is a complex algorithm that relies on a number of auxiliary functions, which, in turn, are also described operationally and often rely on other auxiliary functions themselves.

We explain the DOM `Dispatch` algorithm via an example given in Figure 6.4, which shows a DOM tree of an HTML page with an element `dv` containing two buttons, `bt1` and `bt2`, and illustrates the steps taken by `Dispatch` when the user clicks on `bt1`. Coarsely, `Dispatch` first determines the *propagation path* of the triggered event, i.e. the list of DOM nodes connecting the element on which the event was triggered to the root of the DOM document, in this case [`bt1`, `dv`, `bd`, `htm`, `doc`]. Then, it executes the handlers registered along that propagation path during three consecutive phases: **(1)** the

```
interface EventTarget {
  constructor();

  undefined addEventListener(
    DOMString type,
    EventListener? callback,
    optional (AddEventListenerOptions or boolean) options = {}
  );
  undefined removeEventListener(
    DOMString type,
    EventListener? callback,
    optional (EventListenerOptions or boolean) options = {}
  );
  boolean dispatchEvent(Event event);
};
```

Figure 6.3.: IDL specification of the `EventTarget` interface

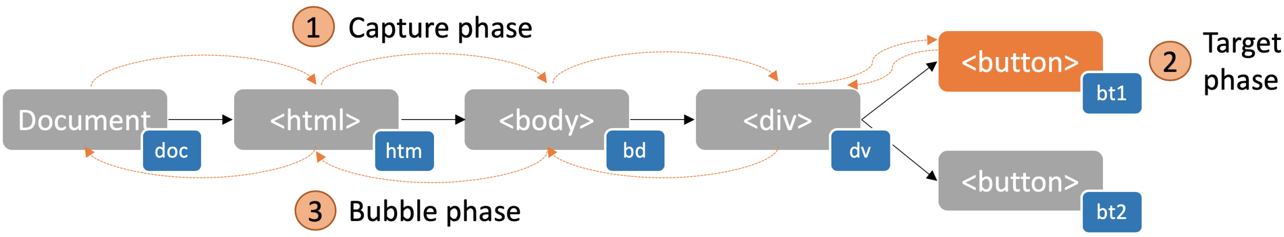

Figure 6.4.: DOM Dispatch Phases

capture phase, where the event is propagated from the root of the document, `doc`, to the target, `bt1`; **(2)** the *target phase*, where the event is processed at the target, `bt1`; and **(3)** the *bubble phase*, where the event is propagated back to the root. During each phase, `Dispatch` executes the handlers attached to the current node if they were registered for the current event and phase. The DOM API method for registering handlers, `addEventListener(type, handler, options)`, allows the programmer to specify if a given handler is to be executed in the capture phase or the bubble phase through the `options` parameter; by default, handlers get executed in the target phase. Importantly, the propagation path is computed only once, before the handlers are executed, meaning that even if their executions alter the propagation path, those changes will not be taken into account by the `Dispatch` algorithm.

6.2.2. API Reference Implementation

We implement all the interfaces defined by the DOM Events API in JavaScript following the standard line-by-line. In Figure 6.5, we show the JavaScript representation of the `EventTarget` and `Event` interfaces. Similarly to the approach used for the implementation of the DOM Core Level 1 API, the fields exposed by each interface are stored in the corresponding objects. For instance, the fields `type` and `eventPhase` are defined in `Event` objects. In contrast, methods exposed by each interface are stored in the respective object prototype. For instance, the methods `addEventListener` and `dispatchEvent` are defined in the `ETProto`, which is the prototype of `EventTarget` objects.

`Dispatch` algorithm implementation. In Figure 6.6, we present our JavaScript (ES5 Strict) implementation of the `Dispatch` algorithm. In the standard, `Dispatch` is presented as a monolithic

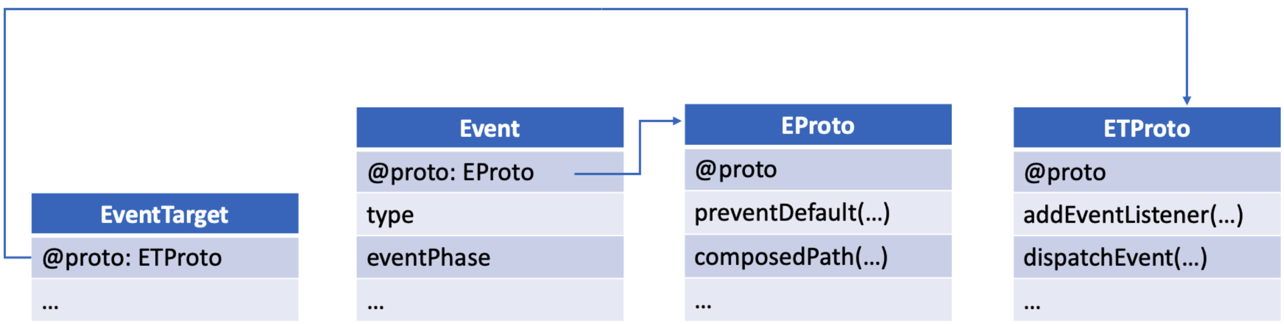

Figure 6.5.: JavaScript object graph for the `EventTarget` and `Event` interfaces

56-line function that is difficult to understand. We instead structure it into seven auxiliary functions, each following the corresponding pseudo-code of the standard line-by-line.

```
function Dispatch(event, target, flags) {
  var relatedTarget = retarget(event.relatedTarget, target);
  var touchTargets = getTouchTargets(event, target);
  var actTarget = isActivationTarget(event);
  updatePropagationPath(event, target, relatedTarget, touchTargets, actTarget);
  captureAndTarget(event, flags)
  if (event.bubbles) { bubble(event, flags) }
  clear(event);
  return !event.canceled
}
```

Figure 6.6.: DOM `Dispatch` implementation

The `Dispatch` algorithm receives as input: the `Event` object that represents the triggered event; the `Node` object on which the event was triggered; and optional flags used to identify a target/event requiring special treatment. [1] The algorithm then proceeds as follows:

1. Call `retarget` to determine the *related target* of the triggered event. Some events are associated with two targets: the main target, supplied as the argument of `Dispatch`; and the related target, determined by `retarget`. For instance, `mouseout`, an event triggered when the user moves the mouse from one node to another, has two targets: the node at which the mouse originally was (main), and the node to which it moved (related).

2. Call `getTouchTargets` to obtain the list of *touch targets* associated with the triggered event. Events involving interactions between the user and a touching surface can be associated with a variable number of targets (e.g., due to the user placing multiple fingers on the surface), called touch targets.

3. Call `isActivationTarget` to check if the event has an associated activation behaviour. For instance, when a `click` event is triggered on a hyperlink, the browser should open a window with the corresponding URL.

4. Call `updatePropagationPath` to the determine the propagation path of the event.

[1] More concretely, the `Dispatch` algorithm receives the flag `legacyTargetOverrideFlag` when the event target is the `Window` object and the flag `legacyOutputDidListenersThrowFlag` when receiving an event originated by the Indexed Database API.

5. Call `captureAndTarget` to execute the capture and target phases.

6. Call `bubble` to execute the bubble phase if the result of inspecting the property `bubbles` of the event object is true.

7. Call `clear` to reset some of the properties of the event object to `null`.

8. Return a boolean indicating if the activation behaviour of the event was not cancelled. When no activation behaviour is defined, the algorithm returns `true`.

DOM Event Model and the JavaScript Semantics. The interaction between the DOM `Dispatch` algorithm and the JavaScript semantics may trigger unexpected behaviours if not properly engineered. Consider, for instance, the following function to be used as a handler:

```
function h(ev) { Object.defineProperty(ev, "bubbles", { get: malicious }) }
```

If the programmer registers `h` as an event handler and that event is triggered, the function `malicious` will be implicitly called when the `Dispatch` algorithm tries to resolve the value of the property `bubbles` after the execution of the target phase, because bubbles is an accessor property (it does not contain a value, but instead getter/setter functions that are executed on property access/update) and `malicious` is its getter. Although the DOM standard defines the `bubbles` property as readonly, the DOM engines of Chrome, Edge, Firefox, and Safari allow it to be re-defined on the event object.[2] Our reference implementation does not suffer from this problem as we define read-only attributes as non-writable on creation.

Connection with the E-semantics. In related works [83, 101], the DOM `Dispatch` is either baked into the formalism, which then becomes complex, and/or not fully faithful to the standard. We take a novel, substantially different approach that allows us both to keep the E-semantics simple and to represent rigorously all of the details of the DOM `Dispatch`. In particular, we store information about DOM handlers directly in their associated `Element` nodes in the JavaScript heap, implement the `Dispatch` fully in JavaScript, and only use the E-semantics to: **(1)** register the `Dispatch` function as the handler of *all* DOM events using the addHdlr primitive; and **(2)** dispatch programmatic DOM events synchronously using the sDispatch primitive. The former effectively means that any time a DOM event (e.g. `click` or `focus`), is triggered, either synchronously or asynchronously, the DOM `Dispatch` function itself is scheduled for execution by the E-semantics. It is then the job of this function, rather than the E-semantics, to traverse the DOM tree and execute the user-register handlers in the appropriate order.

In Figure 6.7, we show our implementation of the `dispatchEvent` function, used to model programmatic dispatch of DOM events. This function calls the E-semantics synchronous dispatch wrapper, `ESem.sDispatch`, in line 4. The behaviour of the **sDispatch** primitive, as given in Chapter 4, precisely captures the programmatic DOM event dispatch as per the standard, where the associated event handlers are meant to be executed immediately.

[2] We discussed this potential vulnerability with one of the members of the committee and they are aware. They made the choice of forbid this kind of scenario only for a few properties assuming the absence of untrusted code.

```
function dispatchEvent(event, flags) {
  if (event.dispatch || !event.initialized) { throw new DOMException(INVALID_STATE_ERR) };
  event.isTrusted = false; event.target = this;
  return ESem.sDispatch(event, this, flags)
}
```

Figure 6.7.: DOM `dispatchEvent` function

Line-by-Line Closeness. We demonstrate that our JavaScript implementation follows the DOM UI Events standard line-by-line by appealing to the code of the `innerInvoke` function, given in Figure 6.8. The `innerInvoke` function is one of the auxiliary functions used by the `Dispatch` algorithm. It is used to execute the listeners for a given event during all three phases of the `Dispatch` algorithm. We illustrate the line-by-line closeness by inlining in comments, for each line of code, its corresponding line in the standard.

```
function innerInvoke (event, listeners, phase, legacyOutputDidListenersThrowFlag) {
  var found = false;                            // 1. Let found be false.
  for (var i = 0; i < listeners.length; i++) {  // 2. For each listener in listeners...
    if (listener.removed) continue;             // ...whose removed is false:
    // 2.1. If event's type attribute value is not listener's type, then continue.
    if (event.type !== listener.type) continue;
    // 2.2. Set found to true.
    found = true;
    // 2.3. If phase is "capturing" and listener's capture is false, then continue.
    if ((phase === "capturing") && (listener.capture === false)) continue;
    // 2.4. If phase is "bubbling" and listener's capture is true, then continue.
    if ((phase === "bubbling") && (listener.capture === true)) continue;
    // 2.5. If listener's once is true, then remove listener from event's currentTarget attribute
    ↪  value's event listener list.
    if (listener.once === true) event.currentTarget.removeListener(listener);
    ...
    // 2.10. Call a user object's operation with listener's callback, "handleEvent", event, and
    ↪  event's currentTarget attribute value.
    execCallBack(listener.handleEvent, "handleEvent", event, event.currentTarget);
    ...
    // 2.13. If event's stop immediate propagation flag is set, then return found.
    if (event.stopImmediatePropagation === true) return found;
  }
  return found; // 3. Return found
}
```

Figure 6.8.: JavaScript implementation of the `innerInvoke` function

6.3. JavaScript Promises API

Promises were introduced into JavaScript (JS) in the 6th version of the standard [28], in response to the increasing popularity and usefulness of various, often incompatible, custom-made libraries for asynchronous computation. Their addition provided clarity and security to JS developers; in fact, the official Promises API has greatly simplified the creation, combination, and chaining of asynchronous computations, eliminating the so-called *callback hell* of multiple nested callbacks [37], which is extremely difficult to understand and reason about.

6.3.1. API Overview

A JS Promise, in essence, is the reification of an asynchronous computation that was either already *settled* in the past or still remains to be settled in the future. A promise can be settled successfully, in which case we say that it is *resolved* (the standard also uses the term *fulfilled*), or unsuccessfully, in which case we say that it is *rejected*. If a promise has not been yet settled, we say that it is *pending*.

At the core of the Promises API is the promise constructor, `Promise`, which is used for creating new promises. This constructor receives as input an *executor function*, which captures the computation to be performed asynchronously. Executor functions have two arguments: a function `resolve` for stating that the corresponding promise has been resolved, and a function `reject` for stating that it has been rejected. Until one of these functions is called, the corresponding promise is left pending.

```
function f(v) { console.log(v) };
var p = new Promise((resolve, reject) => {
  document.getElementById("dv").addEventListener("click",  () => { resolve(1) })
});
p.then(f); console.log(2)
```

Figure 6.9.: JavaScript program calling the `Promise` constructor

Consider the example in Figure 6.9. This program creates a promise `p`, whose executor function registers the function that resolves the promise as the handler for the `click` event on the DOM element with identifier `dv`. This means that `p` will only get resolved after the user clicks on that DOM element. Afterwards, the program uses the `then` function of the Promises API to register a *fulfill reaction* on the promise `p`, meaning that when/if `p` gets resolved, the function `f` will be scheduled for execution with the argument with which `p` was resolved (in this case, `1`). Reactions are scheduled in a *first-in-first-out* manner every time the current computation terminates or yields control. Hence, the program above will always output the string `21` to the console, regardless of how quickly the user is able to click on the DOM element in question.

Note that the example given in Figure 6.9 relies on both DOM and JS Promises. Existing analysis tools for the DOM [72, 84, 3] and JS Promises [36, 66] would not be able to analyse such example because they target either one API or another. One of the advantages of JaVerT.Click is that it enables the analysis of Web programs calling multiple APIs, such as DOM and JS Promises.

Besides the constructor `Promise` and the method `then`, the Promises API provides several other functions for creating, combining, and chaining promises together. The behaviour of these functions/methods is thoroughly described in the ECMAScript standard in pseudo-code. This pseudo-code relies on numerous JavaScript *internal functions* such as `GetValue`, `PutValue` `GetOwnProperty` and `HasProperty`, whose definitions in the ECMAScript standard are also operational, intricate, and intertwined.

6.3.2. API Reference Implementation

We provide a JavaScript reference implementation of the JS Promises API which follows the standard line-by-line. We illustrate, in Figure 6.10, the object graph associated with the promise `p` of the example given in Figure 6.9 after the execution of the `then` method, but before the promise gets settled. Each Promise object keeps track of its current state, reactions to be triggered when the

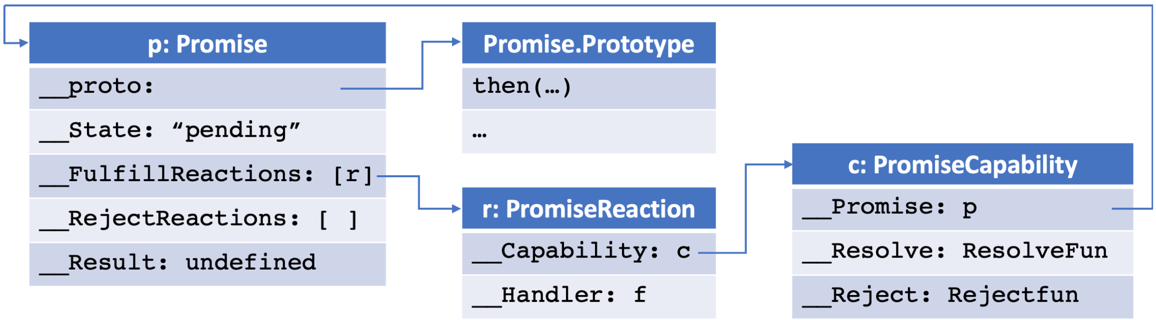

Figure 6.10.: Promises Object Graph

promise is resolved/rejected, and its result, in its internal properties `__State`, `__FulfillReactions`, `__RejectReactions`, and `__Result`, respectively. In this case, the promise `p` is in the "`pending`" state and its result is `undefined`, as it has not been yet resolved. Observe that `f` is registered to execute after `p` using the `then` function in the example; it is not stored directly as a fulfil reaction. Instead, there is a *promise reaction*, `r`, which, in addition to keeping track of `f` in its `__Handler` property, also holds, in its `__Capability` field, a *promise capability* `c`, which keeps track of the promise on whose settlement `f` should be executed (`c.__Promise`), and the `resolve` and `reject` functions given to the executor function of that promise (`c.__Resolve` and `c.__Reject`). In the example, the promise capability `c` contains the promise `p` and the internal resolve and reject algorithms of the standard.

Connection with the E-semantics. Our reference implementation of JS Promises interacts with the E-semantics when triggering Promise reactions for a promise that got settled; this is done by the `TriggerPromiseReactions` function. This function is given as input an array of promise reactions and the value with which their corresponding promise was settled (either resolved or rejected). It then iterates over the elements of the array and, for each element, uses the internal function `PromiseReactionJob` to create an anonymous function that will essentially call the handler of the given reaction with the provided value. This anonymous function is then scheduled for execution directly using the schedule primitive of the E-semantics, as highlighted in line 5 of the following code.

```
1 function TriggerPromiseReactions (reactions, argument) {
2   if (!reactions) return undefined;
3   for (var i = 0; i < reactions.length; i++) {
4     var reactionJob = PromiseReactionJob (reactions[i], argument);
5     ESem.schedule(reactionJob);
6   }
7 }
```

Note that the E-semantics **schedule** primitive, as defined in Chapter 4, adds the given handler to the *end* of the continuation queue. This is consistent with the behaviour of JS Promises described in the standard, Section 8.4.1 [29], which states that pending jobs (essentially, the fulfil and reject reactions) are to be added "at the back of the job queue".

Line-by-Line Closeness. We demonstrate that our implementation follows the ECMAScript standard line-by-line by appealing to the `FulFillPromise` function, described in the Section 25.4.1.4

of the standard; we give its implementation in Figure 6.11, annotated with the corresponding lines of the standard. The `FulFillPromise` function is one of the internal functions used by the function `ResolveFun` (shown in Figure 6.10), which, in turn, is used by promise executors to fulfil their associated promises. The function `FulFillPromise` receives a promise together with the value with which it is to be resolved and proceeds as follows: **(1)** sets the internal state of the given promise object appropriately; and **(2)** schedules the promise's fulfil reactions.

```
function FulfillPromise(promise, value) {
  // 1. Assert: The value of promise's [[State]] internal slot is "pending".
  Assert(promise.__State === "pending");
  // 2. Let reactions be the value of promise's [[FulfillReactions]] internal slot.
  var reactions = promise.__FulfillReactions;
  // 3. Set the value of promise's [[Result]] internal slot to value.
  promise.__Result = value;
  // 4. Set the value of promise's [[FulfillReactions]] internal slot to undefined.
  promise.__FulfillReactions = undefined;
  // 5. Set the value of promise's [[RejectReactions]] internal slot to undefined.
  promise.__RejectReactions = undefined;
  // 6. Set the value of promise's [[State]] internal slot to "fulfilled".
  promise.__State = "fulfilled";
  // 7. Return TriggerPromiseReactions(reactions, value).
  return TriggerPromiseReactions (reactions, value)
}
```

Figure 6.11.: `FulfillPromise` function annotated with the corresponding lines of the standard

6.4. async/await API

Promises are often used together with the JS `async/await` API. This API introduces *asynchronous functions*, inside of which one can await on a promise to be fulfilled before proceeding with the current computation. The key point of asynchronous functions is that they *do not block* the execution of their caller function when their execution gets suspended on an `await`; instead, the control is immediately transferred to the caller function, which continues with the execution as if the asynchronous function had simply returned.

Analogously to the DOM reference implementations, the JS async/await API: is implemented directly in JavaScript (ES5 Strict), with the Promises implementation; are thoroughly tested against the latest version of the official ECMAScript test suite [26] (cf. Chapter 7); and make use of their dedicated E-semantics primitives (cf. Chapter 4).

The `async/await` API is defined in the 8th version of the ECMAScript standard [29]; it is meant to be used together, as it is only possible to use `await` inside an asynchronous function. Furthermore, the `async/await` APIs directly build on the Promises API in that they make explicit use of JS Promises functions and methods.

6.4.1. API Overview

In a nutshell, an asynchronous function is a JavaScript function whose execution can *yield*, that is, transfer the control to its calling context without having completed its execution. A call to an asynchronous function is evaluated to a promise that is settled once that function terminates executing: if the function returns, the promise is fulfilled; if the function throws, the promise is rejected. Consider, for instance, the following program:

```
async function f () { if (b === true) { return 1 } else { throw 2 } };
f().then((v) => { console.log(v) }, (v) => { console.log(v) })
```
(CS1)

Recall that the method `then` receives as input two functions which are registered, respectively, as a fulfil reaction and a reject reaction on the `this` object. Hence, the first function is executed if the promise is fulfilled, whereas the second one is executed if it is rejected. Consequently, in the case of the example, if the global variable `b` is equal to `true`, the program will write `1` to the console, otherwise it will write `2`.

As stated above, an asynchronous function can make use of the `await` expression to transfer the control to the calling context. Essentially, the expression (`await e`) evaluates `e` to a promise and suspends the computation of the current function until that promise is settled. Consider, for example, the following program.

```
var p = new Promise(function (resolve, reject) { ... });
async function g () { return await p };
g().then((v) => { console.log(v) }, (v) => { console.log(v) });
```
(CS2)

This time, the asynchronous function `g` awaits on a promise `p`. If/when `p` is settled, `g` returns the value with which it was settled. Suppose, for instance, that `p` is resolved with value `1`; in this case, the function `g` returns `1`, meaning that its associated promise will also be fulfilled with value `1`. Alternatively, suppose that `p` is rejected with value `1`; then, `g` throws `1`, meaning that its associated promise will also be rejected with value `1`. In both cases, the program will simply write `1` to the console.

6.4.2. API Reference Implementation

Because the `async` and `await` operators depend on JS Promises, we build our implementation of the JS async/await API on top of our reference implementation of JS Promises. Essentially, we compile both the `async` and `await` operators to ES5 Strict (the ECMAScript version supported by JaVerT). In the following, we introduce the compilation of JS async/await and show how it connects with the event primitives of the E-semantics.

Compiling `async/await` to ES5 Strict. As `async` and `await` fundamentally change the control flow behaviour of the language, they cannot be simply implemented as libraries. Hence, we introduce a pre-compilation step that translates these constructs to ES5 Strict. Expectedly, the compiled programs use the Promise constructor to create the promise associated with the execution of the asynchronous function being compiled. The key case of the compiler, given below, corresponds to the default

$$\textsc{If}\quad \frac{s = \mathsf{if}(e)\{s'\}\mathsf{else}\{s''\}}{\mathcal{C}_a\langle s\rangle \triangleq \mathsf{if}(e)\{\mathcal{C}_a\langle s'\rangle\}\mathsf{else}\{\mathcal{C}_a\langle s''\rangle\}} \qquad \textsc{Throw}\quad \frac{s = \mathsf{throw}\ e}{\mathcal{C}_a\langle s\rangle \triangleq s} \qquad \textsc{Return}\quad \frac{s = \mathsf{return}\ e}{\mathcal{C}_a\langle s\rangle \triangleq \mathsf{resolve}(e);\mathsf{return}}$$

$$\textsc{Await}\quad \frac{\begin{array}{l} s' = \{\ \mathsf{var\_\_aux} = e; \\ \quad \mathsf{\_\_await(getPredicate(\_\_aux))}; \\ \quad \mathsf{if(\_\_aux.\_\_State === \text{“resolved”})}\{ \\ \quad\quad \mathsf{resolve(\_\_aux.\_\_Result); return} \\ \quad \}\mathsf{else}\{ \\ \quad\quad \mathsf{throw\_\_aux.\_\_Result} \\ \quad \} \\ \} \end{array}}{\mathcal{C}_a\langle \mathsf{await}\ e\rangle \triangleq s'}$$

Figure 6.12.: Auxiliary Compiler

translation[3] of asynchronous functions:

$$\begin{array}{l} \mathcal{C}\langle \mathsf{async\ function}(\bar{x})\{s\}\rangle \triangleq \mathbf{function}(\bar{x})\ \{ \\ \qquad \mathsf{return}\ \mathbf{new\ Promise}(\mathbf{function}(\mathbf{resolve}, \mathbf{reject})\ \{ \\ \qquad\quad \mathsf{try}\ \{\mathcal{C}_a\langle s\rangle;\ \mathsf{resolve(undefined)}\}\ \mathsf{catch(e)}\ \{\mathsf{reject(e)};\} \\ \qquad \}) \\ \} \end{array}$$

Essentially, an asynchronous function is compiled to a normal ES5 Strict function that simply creates a promise `p` and returns it. The body of the original function is run inside the executor of the promise. Additionally, we make use of an auxiliary compiler $\mathcal{C}_a$ to rewrite `return` statements inside the body of the original function so that they are replaced by a call to `resolve` followed by an empty return. The auxiliary compiler also compiles the `await` operator.

In Figure 6.12, we define the auxiliary compiler considering four cases: IF, THROW, RETURN and AWAIT. For most cases, the auxiliary compiler does not change the original code; it simply keeps the original structure. This can be observed on the IF and THROW cases. However, in the presence of `async` functions, the auxiliary compiler replaces each `return` statement by a call to `resolve` followed by an empty `return`. The compilation of the `await e` expression is more involved. Concretely, the compiled code first stores the value of `e` in an auxiliary variable `__aux`. Then, it uses the await primitive of the E-semantics with the argument `getPredicate(__aux)`, which corresponds to a function that evaluates to `true` once the promise has been settled. When the await primitive (c.f. Chapter 4) is called, the current execution is suspended until the given predicate holds, meaning that the current function resumes when the promise is either resolved or rejected. Then, the compiled code checks if the promise was fulfilled: if so, it continues with the execution normally; if not, it throws the value with which the promise was rejected. For clarity, the introduced compiler is a simplified version of the `async/await` compilation and it does not follow the standard line-by-line.

In Figures 6.13 and 6.14, we give the compilation of the functions `f` and `g`, given in Code Snippets 1

[3] If an asynchronous function can return from a `finally` block, the settling of its associated promise must be deferred to that `finally` block, making the compilation of `return` statements more complex.

and 2 respectively. Note that we omit the call to resolve(undefined) as, in these particular example, it represents dead code due to the presence of `return` statements inside the `async` functions.

```
function f () {
  return new Promise (function (resolve, reject) {
      try { if (b === true) { resolve(1); return } else { throw 2 } }
        catch (x) { reject(x) }
    })
}
```

Figure 6.13.: Example showing `async` compilation

```
var p = new Promise(function (resolve, reject) { ... });
function g () {
  return new Promise (function (resolve, reject) {
      try {
         var __aux = p;
         __await(getPredicate(__aux));
         if(__aux.__State === "resolved") { resolve(__aux.__Result); return }
         else { throw __aux.__Result }
      } catch (x) { reject(x); }
    })
}
```

Figure 6.14.: Example showing `await` compilation

Line-by-line Closeness. For `async`/`await`, we depart from our line-by-line closeness approach. The reason is that this would require JSIL to support first-order execution contexts, which, in turn, would constitute a considerable engineering effort, including changing the internal representation of execution contexts and extending JSIL with various primitives for their manipulation. Instead, we opted for a more lightweight, compilation-based, approach that still correctly models the `async`/`await` behaviour described in the standard.

6.5. The WebMessaging API

In order to allow for the communication between JS programs included in different windows, the WHATWG group [141] designed the WebMessaging API as part of the HTML5 specification [138]. The communication mechanisms provided by the WebMessaging API were then extended to account for the communication between WebWorkers so that workers could exchange messages with one another.

Communication between workers happens mostly asynchronously, following the message-passing paradigm [19, 65, 69]. Hence, when a worker sends a message to another, it does not get blocked waiting for the reply. Instead, it registers a handler for processing the message reply (if it ever arrives) and proceeds with the current computation. Behind the scenes, that message is simply added to the message queue of the target worker to be processed afterwards.

```
interface MessagePort : EventTarget {
  undefined postMessage(any message, sequence<object> transfer);
  undefined postMessage(any message, optional SerializeOptions options = {});
  undefined start();
  undefined close();
  // event handlers
  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};
```

Figure 6.15.: IDL specification of the `MessagePort` interface

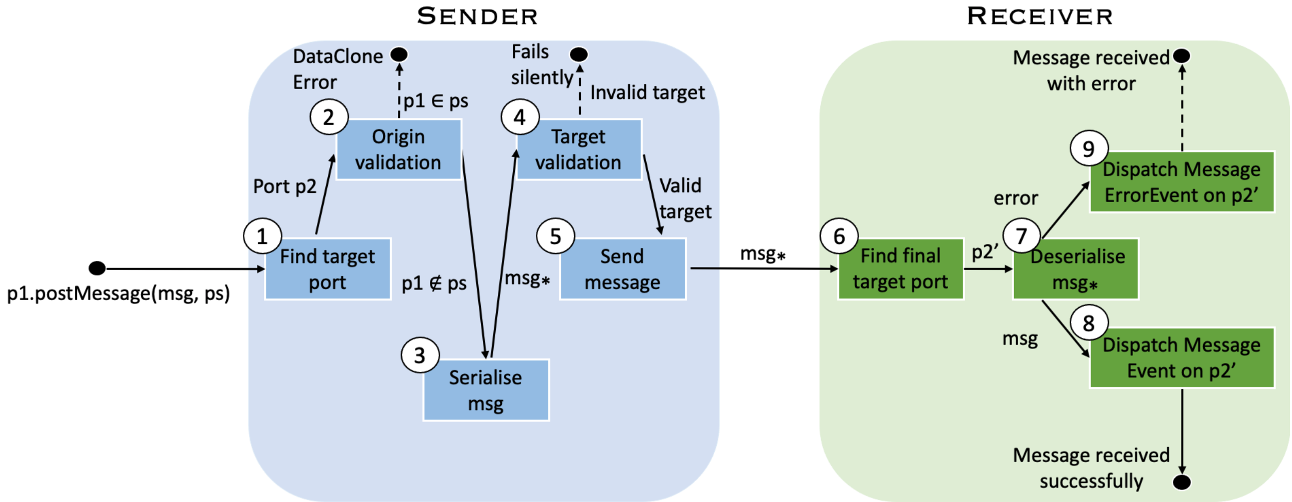

Figure 6.16.: An overview of `MessagePort.postMessage`

6.5.1. API Overview

The WebMessaging API is composed of three main HTML5 interfaces: **(1)** the *Message Channel* interface, which represents a bidirectional communication channel, **(2)** the *Message Port* interface, which represents an endpoint of a bidirectional communication channel, and **(3)** the *Broadcast Channel* interface, which represents a many-to-many communication channel. Unsurprisingly, every message channel is composed of two message ports, corresponding to its two endpoints; these ports are internally connected to each other so that the messages sent through one arrive at the other. In contrast, a broadcast channel is uniquely composed of the corresponding channel name. Both workers and window objects can subscribe to a broadcast channel so when a message is sent through that channel, it gets delivered to all of its subscribers.

Similarly to the DOM Living standard, the structure of all HTML5 interfaces is described in the standard in object-oriented style. For each interface, the standard specifies its exposed fields and methods. Methods are then further described in pseudo-code style with all of their operations being detailed in a step-by-step fashion. As an example, consider the IDL specification of the `MessagePort` interface given in Figure 6.15.

The standard states that every `MessagePort` is also an `EventTarget`, meaning that all message ports must expose the methods and fields described in the `EventTarget` interface. Additionally, every `MessagePort` object exposes:

- the methods `postMessage`, `start` and `close` for sending a message through the channel associated with the port and activating and deactivating the port; and

- the fields `onmessage` and `onmessageerror` for storing the handlers to be executed when a message gets delivered to the port and when a communication error occurs.

The `postMessage` method is of special interest to us as it lies at the core of the WebMessaging API. In fact, several HTML5 interfaces, such as `MessagePort`, `Window` and `Worker`, include their own version of `postMessage`. Here, we only describe the `postMessage` method of message ports. The others are analogous. Importantly, the `postMessage` method is used to transfer both data and ports. Hence, the execution of `p1.postMessage(msg, ps)` will cause the message `msg` to be sent from `p1` to its associated target port **and** the list of ports `ps` to be transferred from the execution context of `p1` to that of its target port.

Figure 6.16 describes the main steps that are performed when calling `postMessage` on a message port, illustrating both the sender (left) and receiver (right) perspectives. The first step of the algorithm is to find the target port associated with the provided port `p1` (step 1), which we denote by p2. Next, it checks if the origin port `p1` is being transferred (step 2), i.e. if `p1` is included in the list of ports ps to be transferred. If this is the case, a DataCloneError is raised. Otherwise, the algorithm proceeds to the message serialisation step (step 3). The serialisation mechanism involves several sub-steps that we do not detail here, simply using msg_* to denote the serialisation of the input message msg. For instance, the JavaScript object {conference: "ECOOP", year: 2022} is serialised as:

{Type: "Object", Properties: [{Key: "conference", Value: {Type: "primitive", Value: "ECOOP"},
{Key: "year", Value: {Type: "primitive", Value: 2022}}]}

Then, the algorithm validates the target port (step 4). If port p2 has value null or is being transferred (meaning that it belongs to ps), the algorithm fails silently and no message is sent. Otherwise, if port p2 is valid, the serialised message is sent from p1 to p2 (step5).

On the receiver side, the initial step is to find the *final target port* p2' (step 6). Typically, the final target port coincides with the one found on the sender side (in our case, p2 = p2'). However, if the target port has been transferred before the message gets delivered, the message may need to be re-directed to current execution context of the target port. Next, the message is deserialised (step 7). Finally, if the deserialisation succeeds, a DOM MessageEvent is dispatched on port p2' with the de-serialised message as an argument. Otherwise, a MessageErrorEvent is dispatched on p2'. Note that the WebMessaging API relies on the DOM Events API for notifying message targets, such as message ports, of their incoming messages. This was one of our motivations for designing the MP-semantics parametrically on the E-semantics.

6.5.2. API Reference Implementation

Our reference implementation WebMessaging covers the interfaces `MessagePort`, `MessageChannel`, and `BroadcastChannel`, all specified in Section 9.5 of the HTML5 standard. Additionally, our implementation partially supports cross-document messaging (Section 9.4.3) and message broadcasting between different browsing contexts (Section 9.6). Cross-document messaging is typically used to

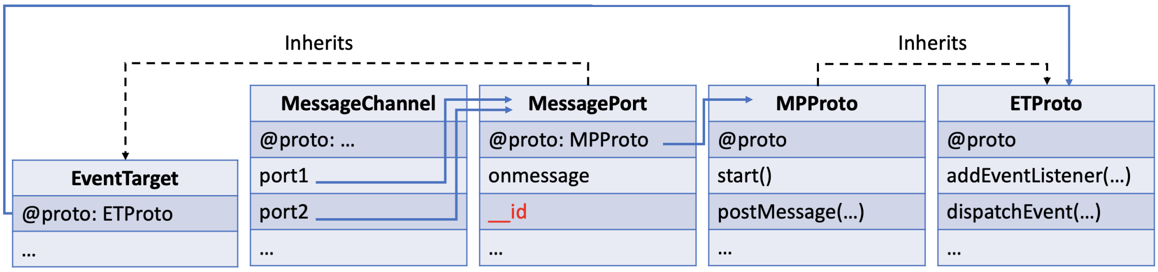

Figure 6.17.: JavaScript object graph for the `MessagePort` interface

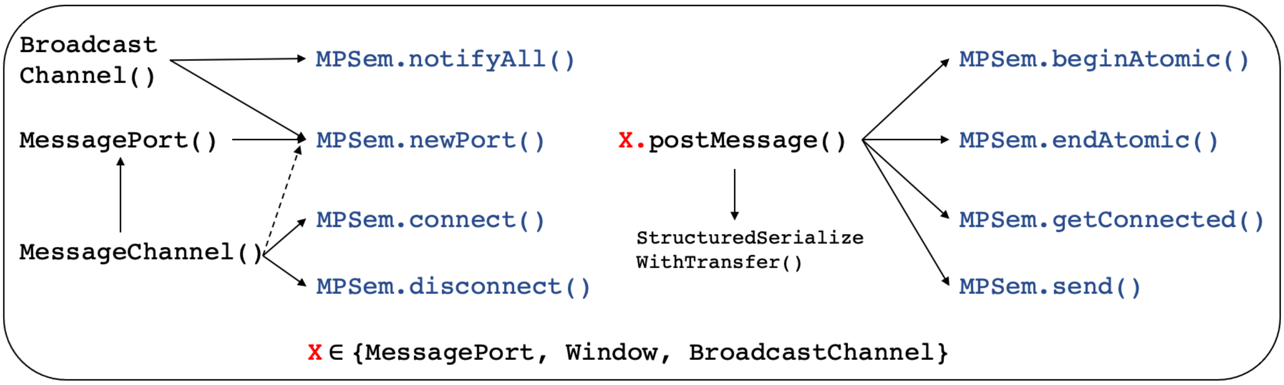

Figure 6.18.: Fragment of the call graph of our WebMessaging reference implementation

enable communication between different `Window` objects through the use of the `Window` dedicated `postMessage` method; e.g. `Window.postMessage`(). Importantly, communication between `Window` objects is treated differently depending on whether or not the two `Window` objects belong to the same origin [93]. So far, our implementation only supports same-origin communication. We believe that its extension to allow for cross-origin communication is straightforward, not requiring any modifications to the underlying MP-semantics.

We implement the WebMessaging API in JavaScript (ES5 Strict), mapping each interface to a JavaScript constructor function in charge of creating the objects of that interface. In Figure 6.17, we show a fragment of the object graph corresponding to our implementation of the interfaces `MessageChannel` and `MessagePort`. Analogously to the approach used for the DOM and JS Promises, we use prototype-based inheritance to emulate standard class-based inheritance, storing the methods shared by all the objects of a given interface in its associated `prototype`. Accordingly, the methods of the `MessagePort` interface are defined in the object `MPProto`. Given that the interface `MessagePort` is supposed to be an extension of `EventTarget`, we define the internal prototype of `MPProto` to be the `ETProto` object, that is, the prototype of all event target objects. In this way, all `MessagePort` objects have access to the methods exposed by `EventTarget` objects, most notably the methods `addEventListener` and `dispatchEvent` shown in the figure.

Connection with MP-semantics. Our reference implementation of WebMessaging relies on the MP-semantics to implement the core message-passing functionality. However, only a few methods of the WebMessaging API directly interact with the MP-semantics. These are shown in Figure 6.18 together with the MP-semantics primitives that they use. In the following, we go through each of these methods explaining how they use their corresponding primitives.

- `postMessage`: Unsurprisingly, the `postMessage` method uses the getConnected⟨⟩ primitive to obtain the identifier of the destination port (i.e. the port to which the origin port is connected) and the primitive send⟨⟩ to send the serialised message to that port. Importantly, the standard mandates that the internal steps triggered by `postMessage` on the sender side and on the receiver side be executed atomically. To achieve this, our implementation makes use of the primitives beginAtomic and endAtomic, which guarantee that no interleaving occurs during the execution of the internal `postMessage` steps.

- `MessagPort`: Whenever a message port is created, our implementation uses the MP-semantics primitive newPort⟨⟩ for creating a new port identifier and assigns this identifier to the property `__id` of the corresponding `MessagPort` object.

- `MessageChannel`: Each message channel is composed of two message ports that must be connected to each other. To this end, the `MessageChannel` constructor first calls the disconnect⟨⟩ primitive on the two ports to guarantee that they are unpaired and then connects the two together using the connect⟨⟩ primitive.

- `BroadcastChannel`: Broadcast channels allow for a many-to-many communication between workers in that each time one of the subscribers of a broadcast channel sends a message on the channel, that message gets delivered to all of the channel's subscribers. Whenever a broadcast channel is created, we assign it an identifier by calling the newPort⟨⟩ primitive. Then, the origin thread notifies all other threads about the creation of a new broadcast channel by calling the notifyAll⟨⟩ primitive. The broadcast channels with the same name of the newly created broadcast channel are then connected to it via the connect⟨⟩ primitive, enabling many-to-many communication.

Line-by-line closeness. Figure 6.19 illustrates the line-by-line closeness between our reference implementation and the standard, showing a fragment of the the `postMessageSteps` function, which is an auxiliary routine of the `postMessage` algorithm that describes the steps to be executed on the sender side, before the message is sent to the target port, as described in Figure 6.16. We include the steps defined by the standard as comments in the code, so that the line-by-line closeness is clear. We highlight the last line of code as it make uses of the primitive send⟨⟩ of the MP-semantics. The `MPSem` object stores wrapper JS functions that trigger the corresponding MP-semantics primitives. Thus, the call to `send`() will trigger the send⟨⟩ primitive of the MP-semantics.

6.6. The WebWorkers API

The first version of the WebWorkers API [133] was published in 2009 by W3C [134] and evolved over the years, eventually being integrated into the HTML5 standard [138]. Web workers represent a radical change to the single-threaded browser execution model, enabling the multi-threaded execution of JavaScript programs. Each worker can be thought of as a separate thread with its own memory. Workers communicate asynchronously using mechanisms described in the WebMessaging API. In the following, we give an overview of the WebWorkers API (§6.6.1) and introduce our reference implementation (§6.6.2).

```
function postMessageSteps (senderPortId, targetPortId, message, options) {
  // 1. Let transfer be options["transfer"].
  var transfer = options["transfer"];
  // 2. If transfer contains this MessagePort, then throw a "DataCloneError".
  if(transfer && transfer.indexOf(senderPortId) !== -1) throw new DataCloneError();
  // 3. Let doomed be false.
  var doomed = false;
  // 4. If targetPort is not null and transfer contains targetPort, set doomed to true.
  var transferIds = transfer.map(function(p){return p.__id});
  if(targetPortId !== -1 && transfer && transferIds.indexOf(targetPortId) !== -1)
     doomed = true;
  // 5. Let serialized be StructuredSerializeWithTransfer(message, transfer).
  var serialized = StructuredSerializeWithTransfer(message, transfer);
  // 6. If targetPort is null, or if doomed is true, then return.
  if(targetPortId === -1 || doomed === true) return;
  // 7. Add a task that runs these steps to the port message queue of targetPort ...
  MPSem.send(serialized, transferIds, senderPortId, targetPortId)
}
```

Figure 6.19.: Line-by-line closeness of the WebMessaging standard and our reference implementation

6.6.1. API Overview

The WebWorkers API defines four main interfaces: **(1)** the `Worker` interface, which represents a *dedicated worker* that is accessible by a single script, **(2)** the `SharedWorker` interface, which represents a worker that can be shared among multiple scripts, **(3)** the `DedicatedWorkerGlobalScope` interface, which defines the fields and methods that are accessible by each dedicated worker and **(4)** the `SharedWorkerGlobalScope` interface, which works similarly to `DedicatedWorkerGlobalScope` for shared workers.

Analogously to the other supported APIs, the interfaces of the WebWorkers API are described in object-oriented style. We show the `Worker` interface as defined by the standard in Figure 6.20. A *worker object* represents the worker's thread inside the thread that created it. Worker objects implement the interface worker, which inherits from `EventTarget`, meaning that worker objects expose the methods defined in the `EventTarget` interface. In addition to the fields and methods defined by the `EventTarget` interface, the `Worker` interface provides:

- a constructor that takes: **(1)** an URL indicating the location of the worker script, and **(2)** an optional argument of type `WorkerOptions` including, for instance, the worker name;
- the methods `terminate` and `postMessage` for terminating the worker and sending a message to its associated running thread; and
- the fields `onmessage` and `onmessageerror`, representing the handlers for the `message` and `messageerror` events.

Every `Worker` object has an implicit `MessagePort` connecting the thread of the script that created the worker to the worker's own thread implicit `MessagePort`. Hence, calling the `postMessage` method on a worker object produces the same effect as calling `postMessage` on its internal `MessagePort`. In Figure 6.21, we show a client of the WebWorkers API containing the main script (left) and two worker scripts: `w1` (center) and `w2` (right). Each script executes in a separate thread. The main script creates

```
interface Worker : EventTarget {
  constructor(USVString scriptURL, optional WorkerOptions options = {});
  undefined terminate();

  undefined postMessage(any message, sequence<object> transfer);
  undefined postMessage(any message, optional SerializeOptions options = {});

  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};
```

Figure 6.20.: IDL specification of the `Worker` interface

```
//Main script
var w1 = new Worker("w1.js");
w1.postMessage("msg1");
```

```
//Worker w1
onmessage = () => {
  postMessage("msg3");
}
var w2 = new Worker("w2.js");
w2.postMessage("msg2");
w2.terminate();
```

```
//Worker w2
onmessage = () => {
  console.log("Message received by w2")
}
```

Figure 6.21.: Example with three scripts: main (left), w1 (center) and w2 (right)

the worker `w1` whose script is located in the file `w1.js`, and sends the message `"msg1"` to `w1`. The worker `w1` defines a handler for the `MessageEvent` by assigning it to the global variable `onmessage`. Hence, every message sent from the main thread to `w1` triggers the `onmessage` handler, which sends the message `"msg3"` back to the main thread. The worker `w1` also performs the following steps: **(1)** creates the worker `w2`, **(2)** sends the message `"msg2"` to `w2`, and **(3)** terminates `w2`. The worker `w2`, in turn, defines an `onmessage` handler, which prints the message `"Message received by w2"` to the console.

The execution of the program shown in Figure 6.21 depends on the scheduling policy used. The WebWorkers API does not define a scheduling policy to regulate how the execution of the main thread is to be interleaved with that of the created workers; browsers are free to implement the scheduling policy that they see fit. In Figure 6.22, we show two possible thread interleavings `I1` (top) and `I2` (bottom) based on different scheduling policies: the former prioritises configuration steps over the processing of messages (`Confs > Msgs`), while the latter prioritises the processing of messages over configuration steps (`Msgs > Confs`). We use different colours to indicate which thread is running at each point in time.

The thread interleaving `I1` includes the following steps:

1. The main thread creates the worker `w1` by invoking the `Worker` constructor. At this stage, the main thread and the worker thread `w1` start executing in parallel;

2. Worker `w1` assigns the `onmessage` handler and creates the worker `w2`. We then have three executing threads: the main thread and the two workers `w1` and `w2`;

3. Worker `w2` executes up to completion, meaning that the `onmessage` handler is assigned;

4. The main thread sends the message `"msg1"` to `w1`;

5. Worker `w1` sends the message `"msg2"` to `w2`;

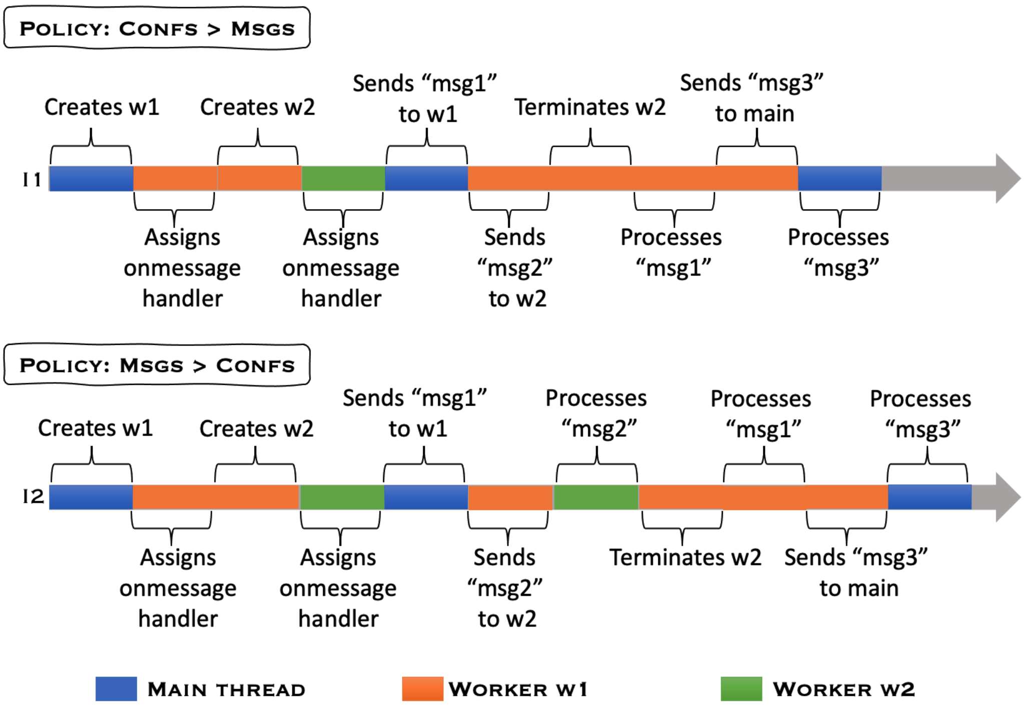

Figure 6.22.: Different scheduling policies for the example shown in Figure 6.21

6. Worker `w1` terminates `w2` by calling `w2.terminate()`. Hence, the worker thread `w2` terminates immediately and its pending message `"msg2"` is discarded;

7. At this stage, both the main and worker scripts finished to execute, and, according to the scheduling policy used, messages can now be processed. Then, `w1` processes message `"msg1"`, and consequently sends message `"msg3"` back to the main thread.

8. Finally, the main thread processes `"msg3"` and the execution terminates as there is neither configuration step nor message pending.

The thread interleaving `I2` is analogous but follows the opposite scheduling policy: the processing of messages is prioritised over configuration steps. This avoids the message `"msg2"` to be discarded. Instead, it is processed before `w1` terminates `w2`. The scheduling policy implemented by major Web browsers prioritises configuration steps over the processing of messages, leaving the processing of messages to be done only if there is no configuration step pending [67, 22]. However, the scheduling policy could vary depending on the reference implementation used. The message-passing module of JaVerT.Click is parametric on a scheduler, allowing for the analysis of the same program using different scheduling policies. The developer just needs to provide the implementation of the corresponding schedulers without the need to modify JaVerT.Click.

6.6.2. API Reference Implementation

We provide a reference implementation of the WebWorkers API including the interfaces `Worker`, `SharedWorker`, `DedicatedWorkerGlobalScope` and `SharedWorkerGlobalScope`, all defined in Section 10.2 of the HTML5 standard. Besides these interfaces, we partially implement various other auxiliar

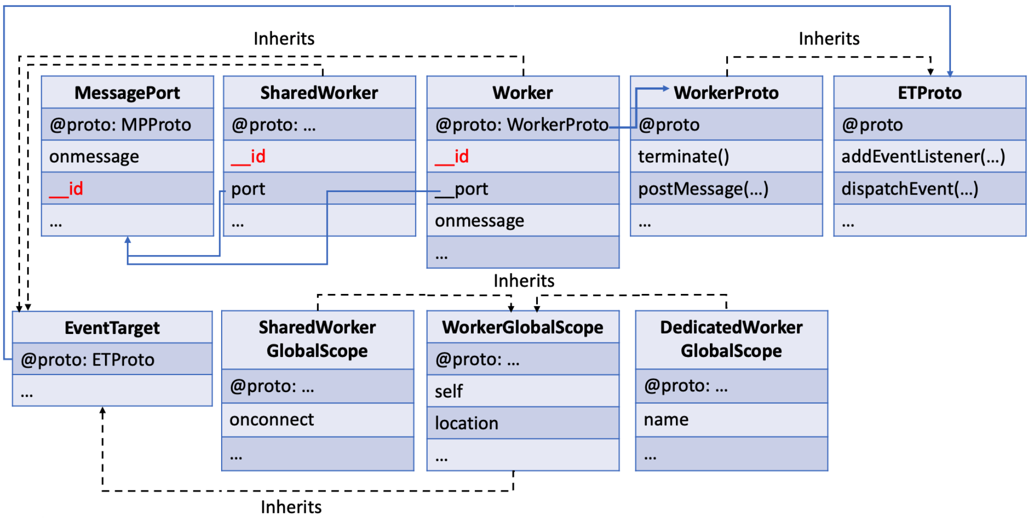

Figure 6.23.: Fragment of `WebWorkers` JavaScript object graph

interfaces, such as the `WorkerNavigator` and `WorkerLocation` interfaces defined in Section 10.3 of the standard.

We design our reference implementation of WebWorkers analogously to the other supported APIs. Each interface from the WebWorkers standard is mapped to a JavaScript constructor. In Figure 6.23, we show a fragment of the object graph for WebWorkers. Similarly to the approach used for the `MessagePort` interface, we also assign an internal identifier `__id` to each `Worker` and `SharedWorker`. Because the `Worker` and `SharedWorker` interfaces inherit from `EventTarget`, their internal prototype objects inherit from `ETProto`, which is the `EventTarget` prototype. The `DedicatedWorkerGlobalScope` and `SharedWorkerGlobalScope` interfaces are specialisations of `WorkerGlobalScope`, which exposes their common fields. The `SharedWorkerGlobalScope` interface exposes the `onconnect` handler to allow multiple scripts to connect to a `SharedWorker`. The `DedicatedWorkerGlobalScope` interface, in contrast, does not expose the `onconnect` field, simply defining an optional `name` property that is mostly used for debugging purposes.

Connection with MP-semantics. The WebWorkers API builds on top of WebMessaging. Hence, it indirectly uses all primitives of the MP-semantics listed in §6.5.2. Besides all those primitives used by the WebMessaging API, our reference implementation of WebWorkers calls the create⟨⟩ and terminate⟨⟩ primitives to create and terminate a `Worker` thread. The primitive create⟨⟩ is called by the constructors of the `Worker` and `SharedWorker` interfaces to setup a new thread, and returns a unique identifier, which is assigned to the internal field `__id` of the respective worker. The primitive terminate⟨⟩ is called by the `terminate` method exposed in both the `Worker` and `SharedWorker` interfaces.

Line-by-line Closeness. Our reference implementation of WebWorkers follows the API standard line-by-line. In Figure 6.24, we show a fragment of the `runWorker` function, which implements the required setup steps for a newly created worker thread. We include the steps defined by the standard as comments in the code, so that the line-by-line closeness is clear. One of the steps consists of establishing bi-directional communication between the new thread and the origin thread, which is

```
function runWorker(worker, workerURL, outsideSettings, outsidePort, options) {
  ...
  //15. Let inside port be a new MessagePort object in inside settings's Realm.
  var insidePort = new MessagePort();
  //16. Associate inside port with worker global scope.
  global.__port = insidePort;
  //17. Entangle outside port and inside port.
  MPSem.disconnect(outsidePort.__id);
  MPSem.disconnect(insidePort.__id);
  MPSem.connect(outsidePort.__id, insidePort.__id);
  ...
}
```

Figure 6.24.: Fragment of `runWorker` function

done through the use of message ports. Given an `outsidePort`, which denotes the port from the origin thread, we need to create an `insidePort`, which denotes the port that is the communication entry point of the new worker thread. We pair the two ports by calling the corresponding primitives of the MP-semantics (highlighted in blue). Finally, after the connection between ports is established, the worker script can start running in parallel with the main thread. The standard does not define a specific scheduling strategy for WebWorkers, but most browser implementations seem to run the newly created worker script up to completion [67, 22].

7. Evaluation

We evaluate JaVerT.Click from two different perspectives. First, we test our reference implementations of DOM Core Level 1, DOM Events, JS Promises, JS async/await, WebMessaging and WebWorkers against their official test suites [57, 26] to make sure that they pass all applicable tests. Second, we evaluate our symbolic execution engine against three open-source libraries: `cash` [142], `p−map` [120] and `webworker-promise` [105].

During the testing process of our reference implementations, we discovered coverage gaps in the DOM Core Level 1 and DOM Events test suites and create additional tests to complete their coverage. Additionally, there were tests of the WebMessaging and WebWorkers test suites that were not consistent with the behaviour described in the HTML5 standard; we reported and fixed these inconsistencies via pull requests to the official test suite repository. All submitted pull requests have been accepted by the HTML5 committee and are now integrated into the official test suite repository.

To evaluate the symbolic engine of JaVerT.Click, we developed a symbolic test suite for each targeted library. The symbolic test suites developed for the open-source libraries revealed, in total, six previously unknown bugs. All bugs have been reported and two have been fixed via pull requests. The remaining four bugs have also been acknowledged by the libraries' developers. By symbolically testing the three libraries, we establish the bounded correctness of several of their functional properties.

Outline. We first detail the testing of the reference implementations (§7.1). Finally, we present the symbolic testing results for the `cash`, `p−map`, `webworker-promise` libraries (§7.2).

7.1. Testing API Reference Implementations

We tested our reference implementations of the chosen APIs against their official test suites. All these test suites come with their own infrastructures and have tests written in different formats. For this reason, we built a common testing infrastructure that converts the tests of all the considered test suites into a uniform JS format that can be handled by JaVerT.Click.

We evaluated each of the implemented APIs using its official test suite as follows:

- The DOM Core Level 1 implementation was tested against the official DOM Core Level 1 test suite [25];
- The JS Promises and JS async/await implementations were tested against the appropriate subsets of the Test262 test suite, which covers the entire ECMAScript standard [26];
- The DOM Events, WebMessaging and WebWorkers implementations were tested against the appropriate subsets of the Web Platform test suite [57].

In the following, we detail our testing infrastructure (§7.1.1), present the test results for all supported APIs (§7.1.2) and discuss the coverage gaps and issues found in the official tests (§7.1.3).

7.1.1. Testing Infrastructure

We illustrate our testing infrastructure in Figure 7.1. The tests for the JS Promises and JS async/await APIs are written in JavaScript. To run them in JaVerT.Click, we only need to compile the test harness together with the tests. The tests for DOM Events, WebMessaging and WebWorkers, in contrast, are written in HTML and contain JS scripts enclosed by the `<script>` tag. We developed a Python script to isolate the code of each test into a JS file. Then, if the test takes any XML/HTML file as input, we also parse the corresponding input file with the help of the `xml-js` parser [146]. This parser returns a JSON[1] object, which is subsequently converted to a DOM tree. The following steps are identical to the ones used for the JS Promises and JS async/await APIs. Finally, as the DOM Core Level 1 tests are written in XML, we additionally have to first transform them into HTML tests using XSLT, and then apply the same steps used for DOM Events, WebMessaging and WebWorkers. In the following, we provide an example of an XML test to illustrate the transformations applied.

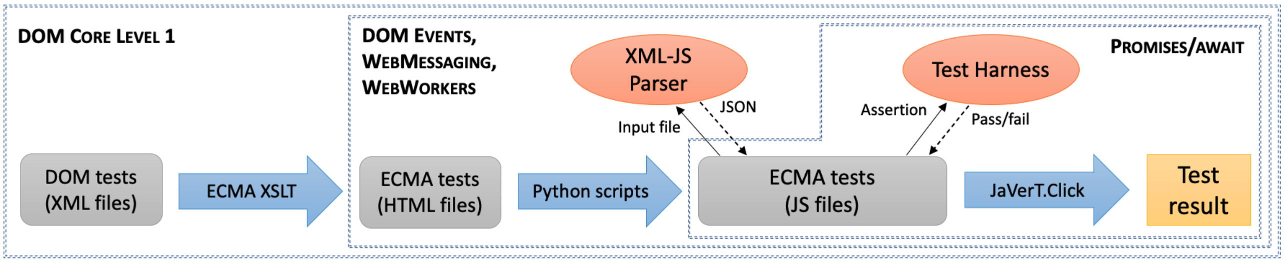

Figure 7.1.: Common Testing Infrastructure

Example. In Figure 7.2, we show an official test[2] written in XML taken from the DOM Core Level 1 test suite (left) and the resulting JavaScript test after applying the transformations (right). In general, each XML tag in the original test becomes a JavaScript command in the transformed test. For instance, each `<var>` XML tag is transformed into a JavaScript variable declaration. Analogously, each `<load>` tag is transformed into a variable assignment. Other tags representing functions provided either by the DOM API or by the test harness, such as `<createEntityRef>` and `<assertNotNull>`, are transformed into JS function calls. Note that this test takes as input the `staff.xml` file. In this particular example, besides applying the ECMA XSLT transformation and using the Python script, our infrastructure also parses the `staff.xml` file by calling the `xml-js` parser and creates the corresponding DOM tree stored in the variable `doc`.

7.1.2. Test Results

In Table 7.1, we summarise the results of the testing of our reference implementations. We provide, for each API: **(1)** the number of available tests, **(2)** the number of filtered tests, **(3)** the number of applicable tests, these being the number of available tests minus the number of filtered tests, and **(4)** the number of passing tests. Note that we pass all the applicable tests for all our API reference implementations. We filter out the tests that require features not supported by JaVerT.Click, except

[1] `https://www.json.org/json-en.html`
[2] `https://dev.w3.org/2001/DOM-Test-Suite/tests/level1/core/attrremovechild1.xml`

```xml
<test xmlns="..." name="attrremovechild1">
...
<title>attrremovechild1</title>
...
<var name="doc" type="Document"/>
<var name="entRef" type="EntityReference"/>
...
<load var="doc" href="staff".../>
<createEntityRef obj="doc" var="entRef".../>
<assertNotNull actual="entRef" id="..."/>
...
</test>
```

```javascript
(function attrremovechild1() {
  var doc;
  var entRef;
  ...
  doc = parser.parseDocument("staff.xml");
  entRef = doc.createEntityReference("ent4");
  assertNotNull("EntRefNotNull",entRef);
  ...
})()
```

Figure 7.2.: XML test taken from test suite (left) and corresponding JavaScript transformed file (right).

of the DOM Core Level 1, for which we pass all available tests. We explain the filtering criteria in more detail later in this section.

	DOM Core Level 1	DOM Events	JS Promises	JS async/await	WebMessaging	WebWorkers
Available Tests	527	83	474	86	121	269
Filtered Tests	0	27	130	18	30	111
Applicable Tests	527	56	344	68	91	158
Passing Tests	**527**	**56**	**344**	**68**	**91**	**158**
Line Coverage	98.14%	97.45%	98.76%	N/A	N/A	N/A
Untested Lines	13	8	5	N/A	N/A	N/A
Additional Tests	5	3	N/A	N/A	N/A	N/A

Table 7.1.: Testing of Reference Implementations

We measured test suite line coverage of the DOM Core Level 1, DOM Events and JS Promises test suites. The three test suites have comprehensive coverage, but there were still gaps that allowed us to develop additional tests. We developed 5 additional tests for the DOM Core Level 1 API and 3 additional tests for the DOM Events API. The 5 untested lines in our implementation of the JS Promises API are part of the `Promise.allSettled` function,[3] which was not part of the standard yet at the time when these tests were executed. Thus, we do not provide complementary tests. We manually checked the filtered tests for DOM Events and JS Promises to make sure that our coverage results are trustworthy. Hence, we believe that including the filtered tests of the DOM Events and JS Promises test suites would not affect our coverage results. We do not provide coverage results for the JS async/await, WebMessaging and WebWorkers test suites because we do not implement all features from these APIs and we filter out a substantial number of tests. Thus, we believe that the coverage results for these three APIs would not be representative enough.

We did not submit the additional tests written for DOM Core Level 1 because the test suite is not used anymore. However, it is the only test suite available for the DOM Core Level 1 and it allowed us to test our reference implementation. Since the DOM standard has evolved substantially over the years, its current test suite [57] covers the entire DOM Living Standard. We reported [52] the coverage gaps of the DOM Events test suite to the committee, and, if possible, the additional tests will be integrated into the test suite repository.

[3] `https://tc39.es/proposal-promise-allSettled/`

	DOM Events	JS Promises	JS async/await	WebMessaging	WebWorkers
404 Not Found Error	-	-	-	-	1
Ajax	2	-	-	-	-
Animation Frame	-	-	-	-	1
Array Buffer	-	-	-	2	2
Atomics	-	-	-	-	1
Blob	-	-	-	5	10
Canvas	-	-	-	3	2
Cross-origin	5	-	-	14	18
CSS	19	-	-	-	-
Dynamic Import	-	-	-	-	20
Error Line Number	-	-	-	-	5
ES6 Features	1	127	18	-	-
IFrame Detach	-	-	-	-	1
Navigation	-	-	-	-	6
NetworkError	-	-	-	-	6
Non-strict Mode	-	3	-	-	-
Opaque Origin	-	-	-	2	-
Python	-	-	-	-	2
RegExp	-	-	-	4	8
Timers	-	-	-	-	2
Termination	-	-	-	-	4
URL Validation	-	-	-	-	7
UTF Encoding	-	-	-	-	2
XMLHttpRequest	-	-	-	-	13
Total	27	130	18	30	111

Table 7.2.: Filtered tests

Test Filtering Criteria. In order to run the official tests against our reference implementations, we filtered tests that use features not supported by JaVerT.Click, such as ES6 features. In Table 7.2, we provide, for each API, the number of tests filtered per category. We filtered out 27 tests from the DOM Events test suite, of which 19 depend on Cascading Style Sheets (`CSS`),[4] which is a language used for defining the style of a webpage. JaVerT.Click does not support `CSS` and this feature is not part of the DOM standard. The remaining 8 tests depend on 404 Not Found Error, Canvas and capturing line numbers of JavaScript errors, which are also not core features of DOM Events.

We filtered out 130 tests from the JS Promises test suite, of which 127 depend on ES6 [28] features and 3 depend on Non-strict mode, which is not supported by JaVerT.Click. Non-supported ES6 features include Symbol iteration, classes, reflection and proxies. All the 18 tests filtered out from the JS async/await test suite also depend on ES6 features, such as default arguments and generators.

We filtered out 30 tests from the WebMessaging test suite, of which 14 depend on cross-origin communication, which allows for the communication between `Window` objects holding HTML documents coming from different origins. The remaining 16 depend on Array Buffer, Blob, Canvas, Opaque Origin and RegExp. All filtering criteria, except cross-origin communication, are orthogonal to the WebMessaging and WebWorkers APIs, being mostly related to features of the more recent versions of the ECMAScript standard [28, 29] that are unsupported by JaVerT. We believe that one could apply our methodology by extending JaVerT with support for these features without changing the MP-semantics module at the core of JaVerT.Click. Finally, we filtered out 111 tests from the WebWorkers test suite, of which 20 depend on dynamic import and 18 depend on cross-origin communication and 13 depend on XMLHttpRequest. The dynamic import and XMLHttpRequest features are not part of the WebWorkers standard and thus not relevant in the context of our MP-semantics. We do not detail the remaining filtered tests for brevity, but their filtering criteria is not related to core features

[4] `https://www.w3.org/Style/CSS/Overview.en.html`

of the WebWorkers API.

7.1.3. Contribution to official test suites.

During the process of testing our reference implementations against the official test suites, we identified coverage gaps and bugs in the various test suites. The incorrect tests had to be fixed because they were either inconsistent with the behaviour described in the standard or contained a bug in their respective implementation. In the following, we provide examples of added and fixed tests.

Added Tests

In order to fill the coverage gaps identified in the DOM test suite, we developed 5 additional tests to the DOM Core Level 1 test suite and 3 additional tests to the DOM Events test suite. In Figure 7.3, we show one of the additional tests developed for the DOM Core Level 1 test suite. The scenario explored in the test is the insertion of a text node (`textNodeChild`) as child of another text node (`textNode`). According to the specification of the DOM Core Level 1 [130], a text node cannot have children. Hence, this test expects the call to `appendChild` to raise a DOM exception of type `INVALID_HIERARCHY_ERR`.

```
(function nodeappendchildinvalidhierarchy() {
   var success;
   var doc;
   var textNode;
   var textNodeChild;

   success = false;
   // DOM tree is obtained from staff.xml file.
   doc = docs["staff.xml"]
   textNode = doc.createTextNode("new text node");
   textNodeChild = doc.createTextNode("child text node");
   try {
       // This call must raise exception
       textNode.appendChild(textNodeChild);
   }
   catch(ex) {
       // Check if exception raised was of type INVALID_HIERARCHY
       success = (typeof(ex.code) != 'undefined' && ex.code == 3);
   }
   // Check that the variable success is set to true
   assertTrue("throw_INVALID_HIERARCHY_ERR",success);
})()
```

Figure 7.3.: DOM Core Level 1 additional test.

Fixed Tests

In Figure 7.4, we show one of the failing tests, in which we found two issues. The goal of this test is to ensure that certain interfaces are globally available inside the running thread of a shared worker. The test includes the worker (left) and main (right) scripts.

Worker script. First, in line 1, the worker script declares the `prt` variable that denotes the port that is used to establish a connection with the main thread. Then, in lines 2-9, the script defines the function `handleCall` that iterates over the `e.data` array, which contains the interfaces that must be present in the global object of the `SharedWorker` script. The interfaces that are found not be present in the global object are added to the `log` array. In line 8, the worker thread sends a string containing the missing interfaces to the main thread. Finally, in lines 10-13, the script defines the `onconnect` function of the shared worker, which is executed whenever the worker is instantiated. In this case, the `onconnect` function simply sets the handler of the worker's implicit port to the `handleCall` function defined in lines 2-9, meaning that this function is used to process the messages that come from the main thread.

```
1  let prt;
2  const handleCall = e => {
3    const log = [];
4    for (let i=0; i < e.data.length; ++i) {
5      if (!(e.data[i] in self))
6        log.push(e.data[i]);
7    }
8    prt.postMessage('These were missing:'+
   ↪   log.join(', '));
9  };
10 onconnect = e => {
11   prt = e.ports[0];
12   prt.onmessage = handleCall;
13 };
```

```
1  async_test(t => {
2    const expected = 'XMLHttpRequest'+ ... +
3    'SharedWorker ApplicationCache'.split(' ');
4    const supported = [];
5    for (let i=0; i < expected.length; ++i) {
6      if (expected[i] in window)
7        supported.push(expected[i]);
8    }
9    const worker = new SharedWorker('worker.js');
10   worker.port.start();
11   worker.port.postMessage(supported);
12   worker.port.onmessage = e => {
13     assert_equals(e.data, '');
14   });
```

Figure 7.4.: Test taken from WebWorkers official test suite including worker (left) and main (right) scripts.[5]

Main script. The main script contains the actual test code. In line 2, the variable `expected` is assigned to a list of interfaces that should be accessible in shared workers. We omit some of these interfaces to improve readability. Next, in lines 4-8 we store the list of interfaces that are accessible through the `window` object in the variable `supported`. Then, in line 9, we create a new `SharedWorker` object giving the filename `worker.js`. In lines 10-11, the worker port is enabled and the main script sends the `supported` array of interfaces to the worker script as a message. Finally, in lines 12-14, the `onmessage` handler of the `worker` port is assigned to a function that checks if the message received from the worker is equal to the empty string, which should be the case if the `SharedWorker` thread has access to the same interfaces accessible by the main thread.

The test has two issues. Note that the check performed by `assert_equals` in line 13 always fails because the string sent from the worker to the main thread is never equal to the empty string, since, even if no interface is found to be missing the worker thread will still send the string `'These were missing:'` to the main thread. The other issue is that both the `SharedWorker` and `ApplicationCache` interfaces must not be accessible in the worker thread, as specified in Section 10.2.6.4 of the HTML5 standard [138]. To fix this test, one would have to remove the strings `'SharedWorker'` and `'ApplicationCache'` from the expected array and modify the `assert_equals` statement in line 13 to the string `'These were missing:'`. We fixed the test by creating a pull request[6] that has been accepted by the committee. We found further issues in three other tests of the WebMessaging API and fixed them via pull re-

[5] `https://github.com/web-platform-tests/wpt/blob/6b6a74e478f36b8039d448d8f44f86033dabefa3/workers/constructors/SharedWorker/interface-objects.html`

[6] `https://github.com/web-platform-tests/wpt/pull/29987`

quests.[7],[8],[9] All pull requests have been merged into the main branch of the repository. The fact that our reference implementations revealed bugs in the official test suite of the HTML5 standard demonstrates their usefulness for the guiding and managing the standardisation process.

7.2. Symbolic Testing of Open-Source Libraries Using JaVerT.Click

We discuss the evaluation of the symbolic testing engine provided by JaVerT.Click which has been tested against three open-source libraries: `cash` [142], `p−map` [120] and `webworker-promise` [105]. We apply the same approach for these three libraries, which consists of developing a symbolic test suite for each library based on the library's concrete test suite. Starting from the concrete tests written by the library developers allows us to identify a set of functional properties satisfied by each library and, based on these properties, write a symbolic test suite. Symbolic tests tend to provide, in general, better coverage with fewer lines of code and stronger correctness results that are beyond the limit of concrete tests. In the following, we introduce each library and present our symbolic testing results.

7.2.1. The `cash` library

The `cash` library [142] is a jQuery[10] alternative for modern browsers that provides jQuery-style syntax for manipulating the DOM. Its main goal is to remain as small as possible, while still staying (mostly) compatible with jQuery and providing its users with a similar set of features. Moreover, it exhibits better performance than jQuery, as it dominantly relies on native browser events rather than on a custom event model. The library has 1,886 lines of JavaScript code and a growing community of users, with more than 21K weekly downloads on NPM[11], and more than 3M overall downloads[12], and more than 5.7K stars on GitHub [142]. We focus our analysis on the `events` module of `cash`, which provides a mechanism for creating and manipulating DOM events, offering additional functionalities and greater level of control with respect to the native DOM event model.

Library Overview

We provide a high-level description of the functions exposed by `cash` library. The `events` module has five main and twelve auxiliary functions. Here, we focus on the main functions, which allow for basic event-related operations, such as registering a handler to an event or dispatching an event.

on(e, h): `ele.on(e, h)` registers the handler `h` for an event `e` on the element `ele`;

off(e, h): `ele.off(e, h)` deregisters the handler `h` for the event `e` on the element `ele`;

one(e, h): `ele.one(e, h)` behaves the same as `.on`, except that `h` can be triggered only once and is automatically deregistered afterwards;

[7] Issue: `https://github.com/web-platform-tests/wpt/issues/29928`;
Pull request: `https://github.com/web-platform-tests/wpt/pull/29988`

[8] Issue: `https://github.com/web-platform-tests/wpt/issues/29546`;
Pull request: `https://github.com/web-platform-tests/wpt/pull/29546`

[9] Issue: `https://github.com/web-platform-tests/wpt/issues/31125`;
Pull request: `https://github.com/web-platform-tests/wpt/pull/31673`

[10] https://jquery.com/

[11] `https://www.npmjs.com/package/cash-dom`

[12] `https://npm-stat.com/charts.html?package=cash-dom&from=2016-01-04&to=2022-01-04`

Table 7.3.: Symbolic Test Suite for the `events` module of `cash`

Test Name	`rHand`	`sHand`	`tOne`	`tOff`	`other`	**Total**
Time	5.54s	144.38s	24.35s	22.87s	42.20s	**239.34s**
Cmds	1,468,907	38,240,506	9,288,337	9,400,471	14,150,893	**72,549,114**

ready(f): `ele.ready(f)` executes the function `f` after ensuring that the entire document content has been loaded successfully;

trigger(e): `ele.trigger(e)` triggers the handlers for an event `e` on the element `ele`.

The `cash` library comes with a concrete test suite, which has 95.52% overall line coverage. The 18 tests for the `events` module contain 288 lines of code. Their line coverage of `on` is 76.92%, of `trigger` is 93.75%, of `ready` is 0% and of the main auxiliary function called by `on` is 81.82%; the remaining functions have 100% coverage.

Bounded Correctness

We create a symbolic test suite for the `events` module of `cash`, with two goals in mind: **(1)** achieving 100% line coverage for all event-related functions; and **(2)** establishing bounded correctness of several essential properties. We achieve both goals using just eight symbolic tests. In Table 7.3, we give, for these tests, their execution time (Time, in seconds) and the number of executed JSIL commands (Cmds). Each test, additionally, has an overhead of 4.454 seconds, 9 lines of code, and 899,390 executed commands due to the setup of the initial heap and auxiliary testing functions. We single out four tests, which capture important properties that the `events` module should respect; the remaining ones are grouped together as `other`, as they offer little additional insight. These four tests are:

rHand: If a handler has been executed, then it must have previously been registered.

sHand: If a single handler `h` has been registered to a given event `e` using `on(e,h)`, then that is the *only* handler that can be executed for that event. This test has revealed two bugs in the events module of `cash`, discussed in detail in §7.2.1.

tOne: If a single handler `h` has been registered to a given event `e` using `one(e,h)`, then that handler can be executed for that event *only once*.

tOff: If a handler `h` registered to an event `e` is deregistered using `off(e,h)`, then that handler can no longer be executed for that event.

The tests establish that these properties hold *for all* events (strings) up to length 20. The bound 20 has been chosen because it is the length of the longest property of the JavaScript initial heap, `propertyIsEnumerable`. The bound can be adjusted in the tests themselves: the running times will be bound-linear for `rHand`, `tOne`, and `tOff`, which use one symbolic event; and bound-quadratic for `sHand`, which uses two.

The obtained results demonstrate that symbolic testing is far superior to concrete testing: our symbolic test suite has greater coverage, 29% fewer lines of code, and, most importantly, provides much stronger correctness guarantees that are beyond the limit of concrete testing.

Discovered Bugs

As part of its effort to remain minimal, the `cash` library, unlike jQuery, does not implement its own event model. Instead, it heavily relies on the DOM event model. However, the semantics of events in `cash` differs from that of DOM events. For example, `cash` enforces that all user-defined focus-related handlers bubble, by *redirecting* handler registration (via `on` or `one`) and deregistration (via `off`) for the '`focus`'/'`blur`' events to '`focusin`'/'`focusout`' instead. The redirection is implemented as follows: any event that is passed to the `on`, `one`, and `off` functions is first processed by the `getEventTypeBubbling` function:

```
function getEventTypeBubbling(e) { return eventsFocus[e] || e }
```

which is intended to substitute '`focus`' by '`focusin`' and '`blur`' by '`focusout`', while keeping other events intact, by indexing the `eventsFocus` object

```
var eventsFocus = { focus: 'focusin', blur: 'focusout' }.
```

with the event `e`. This indexing is meant to return a string, which is then processed using the `split` function exposed by the `String` interface. This implementation, however, causes two subtle bugs, discovered by the `sHand` symbolic test, whose stylised code, with detailed inlined explanations, is given in Figure 7.5.

```
var count = 0, ele = $('.event'); // Initialise counter and target element

function h () { count++ } // Handler counts the number of times it was called

// Create two symbolic events, e1 and e2, of maximum length 20
var e1 = symbStr(20), e2 = symbStr(20);

// Register the handler for e1 on ele, then trigger e2 on ele
ele.on(e1, h); ele.trigger(e2);

Assert(
  // Handler was executed only once, if e1 and e2 were equal and non-empty,
  (count === 1 && e1 === e2 && e1 !== "") ||
  // and was not executed otherwise.
  (count === 0 && (e1 !== e2 || e1 === ""))
);
```

Figure 7.5.: `sHand` symbolic test.

Bug 1: Overlooked Prototype Inheritance. The first set of counter-examples demonstrates that `cash` throws a native JavaScript type error when executing `ele.on(e1, h)` if

$$\texttt{e1} \in \{\texttt{'constructor'}, \texttt{'hasOwnProperty'}, \texttt{'isPrototypeOf'}, \texttt{'propertyIsEnumerable'}, \texttt{'toLocaleString'}, \texttt{'toString'}, \texttt{'valueOf'}\}.$$

Recall that the function `getEventTypeBubbling` indexes the `eventsFocus` object to redirect focus-related events. Indexing objects as key-value maps, however, may return unexpected values, as shown in [108]: e.g., `eventsFocus`['`valueOf`'] returns the function object found at `Object.prototype.valueOf`, as the 'valueOf' property is not in the eventsFocus object itself, but is in its prototype. Then, since that function object has no split property in its prototype chain, the subsequent call to split throws a native JavaScript type error.

Bug 2: Unintended Event Triggering. The second set of counter-examples demonstrates that the final correctness assertion of the sHand symbolic test does not hold if

$$(\texttt{e1}, \texttt{e2}) \in \{(\texttt{'blur'}, \texttt{'blur'}), (\texttt{'focus'}, \texttt{'focus'}), (\texttt{'blur'}, \texttt{'focusout'}), (\texttt{'focus'}, \texttt{'focusin'})\}.$$

In particular, for the first two counter-examples, the handler registered is not executed even though e1 and e2 are equal. In contrast, for the remaining counter-examples, the handler is executed despite e1 and e2 being different. This bug, similarly to Bug 1, is also related to the implementation of the getEventTypeBubbling function, which is called from the on, one, and off functions, but not from the trigger function. Consequently, user-registered handlers for 'focus' and 'blur' can respectively be triggered *only* via 'focusin' and 'focusout' instead. This is admittedly not intended, and it results from the simplification of the corresponding jQuery mechanisms.

The two bugs found have been reported [43, 42] and acknowledged by to the developers of cash and the second has been fixed [41]. The developers of cash decided to not fix the first bug arguing that it would not be observed in practice. Nevertheless, the same type of bug had coincidently been reported for jQuery[13] and fixed by the library developers.

7.2.2. The p−map Library

The p−map library [120] is a small JavaScript library that extends the functionality of JavaScript promises with the ability to concurrently map over pending promises. Despite having 81 lines of JavaScript code, the p−map library has currently more than 18M weekly downloads on NPM[14], almost 3B overall downloads[15] and 808 stars on GitHub [120]. It calls both the JavaScript Promises and JavaScript async/await APIs. We performed symbolic testing of p−map, where we achieved 98.76% line coverage and discovered a bug [46, 44].

Library Overview

The p−map library is small, having 81 lines of code. The library only exposes the pMap function, which relies on a few internal auxiliary functions. We describe the pMap function below.

pMap(input, mapper, options): Returns a promise p that is fulfilled if all promises resulting from calling the mapper function on each element of the input array are fulfilled; otherwise, if any of the promises is rejected, p is also rejected. The options object can have the following properties:

- **concurrency:** Regulates the maximum number of pending promises at each computation step;
- **stopOnError:** If set to true, the pMap function returns once a promise is rejected. Otherwise, if set to false, the pMap function calls the mapper function on all elements of input and

[13] https://github.com/jquery/jquery/issues/3256
[14] https://www.npmjs.com/package/p-map
[15] https://npm-stat.com/charts.html?package=p-map&from=2016-01-04&to=2022-01-04

rejects afterwards with an aggregated error containing all the errors from the rejected promises.

The p−map library comes with a concrete test suite, which has 97.53% overall line coverage, containing 10 tests. We increase the coverage of the p−map concrete test suite by writing one additional test [47], which has already been integrated [45] into the library concrete test suite.

Bounded Correctness

We develop a symbolic test suite targeting the `pMap` function, establishing the bounded correctness of several functional properties, which are listed below.

SerialXConc: If `pMap` is called with an arbitrary input `i` in serial mode (`concurrency` = 1) must produce the same result of calling it concurrently (`concurrency` > 1).

InvalidConc: If `pMap` is called with an arbitrary input `i` and a non-number `concurrency` value must produce a `TypeError`.

InvalidMapper: If `pMap` is called with an arbitrary input `i` and a non-function `mapper` must produce a `TypeError`.

MaxConc: If `pMap` is called with an arbitrary input `i` the `concurrency` option must not allow the number of pending promises to be greater than the specified `concurrency` parameter.

ErrContinue: If `pMap` is called with an arbitrary input `i`, a mapper `m` which throws an error `e`, and the `stopOnError` optional argument set to `false`, the resulting promise needs to be rejected after `m` is called on all elements of `i`. The resulting error must be an aggregate error containing all errors `e` thrown by `m`.

ErrStop: If `pMap` is called with an arbitrary input `i`, a mapper `m` which throws an error `e`, and the `stopOnError` optional argument set to `true`, the resulting promise needs to be rejected immediately the first error `e` is thrown. The resulting error must be `e`.

Analogously to the approach used in the symbolic testing of the `cash` library, we have written one symbolic test for each functional property of the p−map library. The tests establish that these properties hold given that `concurrency` is a symbolic number with value of up to to 5. The bound can be adjusted in the tests themselves, but because there are tests making use of several symbolic variables, changing the bound can affect performance due to the increase in branching. In Table 7.4, we provide the execution time for each symbolic test (in seconds) and the number of executed JSIL commands.

Test	`SerialXConc`	`InvalidConc`	`InvalidMapper`	`MaxConc`	`ErrContinue`	`ErrStop`
Time	15.466s	0.720s	0.882s	8.377s	5.425s	8.173s
Cmds	2,336,055	104,669	115,232	1,474,012	816,099	691,944

Table 7.4.: Symbolic Test Suite for the p−map library

```
1  function next(){
2    if(isRejected){
3      return; // uncovered line
4    }
5    ...
6    try{
7      await mapper(...);
8    }  catch (e) {
9      if (stopOnError) {
10       isRejected = true;
11       reject(e);
12     }
13   }
14   ...
15 }
```

Figure 7.6.: Uncovered line of `pMap` function

The test `SerialXConc` is the slowest as it makes use of four symbolic variables, thus creating more branching. The tests `InvalidConc` and `InvalidMapper`, in contrast, run faster due to the use of fewer symbolic variables and also to the fact that the number of commands is much smaller, as, in both tests, the `pMap` function is expected to raise an error early in its execution.

The concrete test suite of `p−map` did not cover the scenario tested by `InvalidMapper`. We wrote a symbolic test to guarantee that the property holds, and also a possible concrete test to test the same property to fill the coverage gap. The additional test was submitted [47] and has already been integrated [45] into the library code base.

The results demonstrate that symbolic testing is far superior to concrete testing: we achieve 98.76% line coverage with 6 symbolic tests, while the concrete test suite has 10 tests and 97.53% line coverage. Our symbolic test suite gives much stronger correctness guarantees that are beyond the limit of concrete testing with fewer lines of code and better coverage. Our symbolic tests cover 80 lines of the `p−map` library, which has a total of 81 lines of JavaScript code.

Figure 7.6 shows a fragment of the library function that contains the line of code that is not covered by our symbolic test suite; namely line 3 of function `next`. To understand why this is so, we have to take a closer look at the inner workings of the `p−map` library. It is the job of the function `next` to apply the provided `mapper` to a single element of the given array. Initially, the `pMap` function calls the function `next` n times, with n being equal to the minimum between the size of the given array and the concurrency parameter. Each time a promise is resolved, the library calls the function `next` to check if there is still an index of the given array to be processed and, if that is the case, to apply the mapper to that index. The first `if` statement in the code of function `next` guards against the possibility that the global promise, the one corresponding to the entire array of promises, is resolved between the moment when the last array-index promise was resolved and the moment when the function `next` starts executing. This behaviour is, however, not possible according to the semantics of JavaScript promises and asynchronous functions. The reason is simple: in all contexts where the function `next` is called the flag `isRejected` must be false and there is no yielding of control between the call to `next` and the execution of its body, meaning that there is no way for the flag `isRejected` to become true between the moment one calls `next` and the moment it starts executing.

Discovered Bugs

The `pMap` function exposed by the `p−map` library iterates over an `input` array and applies the given `mapper` function to each element found in the array. In Figure 7.7, we show a fraction of the code of the `pMap` function. First, it validates the given input. For instance, the `pMap` function requires the optional argument concurrency to be of type `number` and to be greater or equal to `1`. Then, there is a `for` loop in which the `next` function (previously introduced in Figure 7.6) is called `n` times, where `n` is equal to the value of the optional argument `concurrency`.

```
if (!(typeof concurrency === 'number' && concurrency >= 1))
  throw new TypeError(`Expected \`concurrency\` to be a number from 1 and up`);
...
for (let i = 0; i < concurrency; i++) {
  next();
  ...
}

function next(){
  ...
  await mapper(...);
  ...
}
```

Figure 7.7.: Fraction of the `p−map` library implementation.

This implementation causes a subtle bug, discovered by the `MaxConc` test, whose stylised code, with detailed inlined explanations, is given in Figure 7.8.

```
// Create a symbolic number of min value 1 and max value 5
var c = symbNumb(5);
// Create the input array with 5 promises
var input = [new Promise(...), ..., new Promise(...)];

(async function (){
  var mapper = async function(n, i, pendingPromises){
    // Ensure that the number of pending promises will never exceed c
    Assert(pendingPromises <= c);
  };

  // Create options object with the concurrency property set to c
  var options = {concurrency: c};
  // Call the pMap function passing arguments input, mapper and options
  pMap(input, mapper, options);
})();
```

Figure 7.8.: `MaxConc` symbolic test developed for the `p−map` library.

Bug: JavaScript Dynamic Typing. The following set of counter-examples demonstrates that the correctness assertion of the `MaxConc` test does not hold if

$$\texttt{concurrency} \in \{1.5, 2.5, 3.5, 4.5\}.$$

In particular, the test fails if `concurrency` is a floating point number, such as `1.5`, `2.5`, `3.5` and `4.5`. This happens because **(1)** the `pMap` function requires the value of `concurrency` to be of type number instead of integer and **(2)** the `for` loop inside the `pMap` function (see Figure 7.7) allows the counter `i` to be greater than `concurrency` if `concurrency` is a floating point number.

We reported [46] the bug, which has been fixed [44] by the main developer of the `p−map` library. The fix consists of ensuring that `concurrency` has type integer instead of type number. The developer also added a test to the concrete test suite in order to make sure that the property will not be violated in future.

Note that, in order to symbolically test the `p−map` library, we rely on our ES6+ transpiler, which covers JS Promises, JS async/await and lambda functions. Our transpiler is trustworthy in the sense that we pass all applicable tests from the test262 test suite for the JS Promises and JS async/await APIs. Additionally, the testing of `p−map` helps to establish trust in the transpiler. In contrast to the approach used for JS Promises and JS async/await, we did not test our transpiler against official tests for lambda functions, but this is not a core contribution of our work. However, the compilation of lambda functions was tested as part of the testing of our reference implementations and open-source libraries. During this process, we did not observe any issue related to the compilation of lambda functions.

We could, in future, provide built-in support for additional features of the ECMAScript standard such as classes and methods. This would be possible, for instance, by implementing those features as part of the JS2JSIL runtime provided by JaVerT. However, implementing advanced features in the intermediate language of JaVerT (JSIL) involves a considerable amount of engineering work. Alternatively, we could extend our transpiler with such ES6+ features. This would be a more straightforward approach since the JS2JSIL runtime covers the ES5 strict version of the standard. When extending our transpiler, we would need to be careful with, for instance, the order in which such transformations would be handled by the transpiler to avoid interference. Depending on the complexity of the new features, it could be necessary to configure the transpiler to apply transformations at multiple steps instead of compiling the entire code to ES5 at once.

Despite the `p−map` library being small, its symbolic analysis revealed a previously unknown bug that had not been found with the use of the existing concrete tests. Additionally, we provide bounded correctness of 5 important functional properties. Again, this shows the importance of symbolic tests and how they can complement concrete tests.

7.2.3. `webworker-promise` library

We previously presented the symbolic testing results for the `cash` and `p−map` libraries, which rely on the DOM Core Level 1, DOM Events, JS Promises and JS async/await APIs. Our events model of the event semantics introduced in Chapter 4 can capture the essence of these APIs. In order to test our message-passing semantics introduced in Chapter 5, we perform symbolic analysis on the `webworker-promise` [105] library, which is a promise-wrapper over the WebWorkers and WebMessaging APIs with 8K lines of code on GitHub, 2,190 weekly downloads on NPM[16] and a total of 413,794 downloads.[17] Similarly to the approach used in the symbolic testing of the `cash` and `p−map` libraries, we develop

[16] `https://www.npmjs.com/package/webworker-promise`
[17] `https://npm-stat.com/charts.html?package=webworker-promise&from=2017-01-04&to=2022-01-04`

a symbolic test suite for the `webworker-promise` library, uncovering three previously unknown bugs and providing bounded correctness guarantees of 10 functional properties.

Library Overview

The `webworker-promise` library exposes three modules: **(1)** a module providing basic functionality on top of the WebMessaging and WebWorkers APIs (which we refer to as `Base` module), **(2)** an `EventEmitter` module to allow for event-based operations to be performed across multiple workers, and **(3)** a `WorkerPool` module to allow for the creation of pools of workers. In the following, we describe the three modules and their main functions.

`Base` module. This module allows for the creation of `WebworkerPromise` objects, which represent the result of the asynchronous computation performed by a worker.

- `WebworkerPromise(w)`: creates a worker-promise object given the `Worker` object `w`.
- `postMessage(m)`: posts the message `m` to the worker thread.
- `terminate()`: terminates the worker thread when called from the main thread.

`EventEmitter` module. This module allows for transparent event-based programming across concurrent workers. For instance, using this module, one can emit an event on the main thread and have event processed by the worker thread with a handler registered for that event.

- `on(e, h)`: registers handler `h` for event `e`.
- `off(e, h)`: deregisters handler `h` for event `e`.
- `emit(e, vs)`: emits event `e` with arguments `vs`.

`WorkerPool` module. This module allows for the creation and management of worker pools.

- `WorkerPool(src, maxThreads, maxConcPerWorker)`: creates a worker pool given the script location, `src`; the maximum number of executing threads, `maxThreads`; and the maximum number of pending messages for a given worker, `maxConcPerWorker`.
- `postMessage(m)`: posts the message `m` to the next available worker from the pool.

Bounded Correctness

By developing a symbolic test suite for `webworker-promise`, we establish the bounded correctness of several functional properties, of which the most important are explained below. The developed test suite covers all the main functions of each module of the `webworker-promise` library.

Mirror: If the main thread sends a message `m` to the worker thread using `postMessage(m)` and the worker thread sends this same message back to the main thread, the message received in the main thread must be equal to `m`.

Terminate: If the main thread sends a message `m` to the worker thread after terminating it using `terminate`(), the message must not be delivered to the worker thread.

Error: If the main thread sends a message `m` to the worker thread using `postMessage`(`m`), and the worker thread raises an error when processing the message `m`, the promise associated with the call to `postMessage` must be rejected in the main thread.

EmitOn: If the main thread emits an event `e` with arguments `vs` on the worker thread using `emit(e, vs)`, and the worker thread registers a handler `h` for `e` using `on(e, h)`, the handler `h` must be triggered.

EmitOff: If the main thread emits an event `e` with arguments `vs` on the worker thread using `emit(e, vs)`, and the worker thread has deregistered the handler `h` for `e` using `off(e, h)`, the handler `h` must *not* be triggered.

EmitOnce: If the main thread emits more than one event `e` with arguments `vs` on the worker thread using `emit(e, vs)`, and the worker thread registers a handler `h` for `e` using `once(e, h)`, the handler `h` must be triggered only once.

PSend: If the main thread creates a pool of workers using the `WorkerPool` constructor and sends a message `m` using `postMessage`(`m`), the next free worker must process the message `m`.

PError: If the main thread creates a pool of workers using the `WorkerPool` constructor and sends a message `m` using `postMessage`(`m`), and the worker thread raises an error when processing the message `m`, the promise associated to the communication in the main thread must be rejected.

PLimit: Suppose that the main thread creates a pool of workers using the `WorkerPool` constructor with maximum number of active threads `n`; if the main thread then activates one or more workers in the pool, the number of available workers must remain less than or equal to `n`.

Op: If the main thread asks to execute an operation `op` on the worker thread and the worker thread registers a callback for `op`, the callback should be executed and the main thread should receive the result from the worker thread.

We have written one symbolic test for each property of the library and used JaVerT.Click to run the tests. The tests establish that these properties hold for *all* messages and events (strings) up to length 20. Table 7.5 provides the execution time for each symbolic test and the number of executed JSIL commands. The execution times for the `webworker-promise` library are significantly higher than the ones for the `cash` and `p−map` libraries. This is mainly due to the fact that, when using WebWorkers, the program consumes *at least* twice the memory when compared to standard non-concurrent applications, as each worker has its own separate memory. Unsurprisingly, the test `PoolLimit` takes longer than the others as it creates a pool of workers with a symbolic number of workers, leading to a substantial amount of branching.

Test	`Mirror`	`Terminate`	`Error`	`EmitOn`	`EmitOff`	`EmitOnce`	`PSend`	`PError`	`PLimit`	`Op`
Time	1m32s	0m45s	1m39s	5m33s	5m35s	10m13s	3m8s	2m3s	12m36s	10m34s
Cmds	316,500	151,396	319,608	1,112,574	1,088,356	1,898,784	502,257	377,745	1,722,600	2,012,982

Table 7.5.: Symbolic Test Suite for the `webworker-promise` library

Discovered Bugs

As an outcome of the symbolic testing of `webworker-promise`, we found three previously unknown bugs in the library. We show one of the bugs (Bug 1) and its corresponding symbolic test in §5.1, which verifies the `Mirror` property. We illustrate another bug in the following.

Bug 2: Prototype Inheritance. This bug is analogous to the one found in the symbolic testing of the `cash` library. In Figure 7.9, we show the library code that causes the bug. The `on` function is exposed by the `EventEmitter` module and is responsible for registering a handler `h` to an event `e`.

```
on(e, h) {
  if(!this.__listeners[e])
    this.__listeners[e] = []; // There are no listeners registered for the given event
  this.__listeners[e].push(h); // Add handler h to array of listeners
  return this;
}
```

Figure 7.9.: Function `on` exposed by the `EventEmitter` module of the `webworker-promise` library.

We now show, in Figure 7.10, the `EmitOn` symbolic test which captured the bug, including its main (left) and worker (right) scripts. In summary, the test consists of the main thread emitting an event to the worker thread, which must have previously registered a handler for the emitted event.

Main script. First, in line 1, the main script declares the variable `input`, which represents data sent in the event emitted to the worker thread. Then, in lines 3-4, it creates the `WebworkerPromise` object which allows to establish communication with the corresponding worker thread. In line 6, the main script declares the `resultPromise` variable, whose value is partially omitted in the code, but corresponds to a promise that is resolved when the worker thread replies to the given event. In line 7, the main script declares the variable `event` and assign it a symbolic value of maximum length 20. Next, in line 9, the main script emits the `event` with data `input` to the worker thread by calling `wp.emit(event, input)`. Finally, in lines 11-13, it ensures that the response `res` from the worker thread is equal to `input`.

```
1  var input = 'input';
2
3  const worker = new Worker('worker.js');
4  const wp = new WebworkerPromise(worker);
5
6  const resultPromise = new Promise(...);
7  var event = symbStr(20);
8
9  wp.emit(event, input);
10
11 resultPromise.then(function(res){
12   Assert(res === input);
13 });
```

```
1  const host =
2    RegisterPromise(async (data, emit) => {
3      ...
4    }
5  );
6
7  var e = symb_string(event);
8
9  host.on(e, function (input) {
10   host.emit(e, input);
11 });
```

Figure 7.10.: `EmitOn` symbolic test including the main (left) and worker (right) scripts.

Worker script. First, in lines 1-5, the worker script declares the variable `host` and assigns it a `RegisterPromise` object, which provides event-based operations such as handler registration/deregistration and event dispatching. Then, in line 7, the worker script assigns the value of the existing symbolic variable `event` to `e`. Finally, in lines 9-11, the worker script registers a handler to event `e` by calling the `on` function on the `host` object. Essentially, whenever the worker thread receives an event from the main thread, it emits it back to the main thread with the same input data.

In JavaScript, every object exposes, by default, a set of functions defined in the `Object` interface, including `toString`, `isPrototypeOf` and `hasOwnProperty`. The `on` function shown in Figure 7.9 computes a map of events `e` to handlers `hs`, stored in `this.__listeners`. Making the event equal to any of the properties defined by the `Object` leads to an issue as accessing property `e` of the object gives us a function instead of a list of handlers. Hence, the call to `push` causes a native JavaScript type error. We identified this error in the library and reported to the developers,[18] who acknowledged the bug and will fix for the release of the next version of the `webworker-promise` library.

Bug 3: Dynamic Typing. This bug is analogous to the bug found during the symbolic testing of the `p−map` library and was captured with the `PoolLimit` test. JavaScript is dynamically typed, which makes programs more error-prone. The `WorkerPool` interface of `webworker-promise` takes a maximum number of threads `maxThreads` as input, so pool should then always have no more workers than `maxThreads`. The property is not valid, however, if `maxThreads` is a floating-point number, as the `WorkerPool` module expects it to be an integer. We reported[19] the bug to the developers and fixed it via a pull request.[20]

The symbolic analysis of the `webworker-promise` library uncovered three previously unknown bugs and allowed for the bounded correctness guarantee of 10 functional properties. Two bugs have already been fixed via pull requests and the other bug has been acknowledged and will be fixed. Note that the three libraries analysed cover all APIs supported by JaVerT.Click and our results of the symbolic testing show that the tool is able to analyse real-world code. In contrast to the `cash` and `p−map` libraries that make use of the DOM Core Level 1, DOM Events, JS Promises and JS async/await APIs, the `webworker-promise` library highly depend on the WebMessaging and WebWorkers APIs.

Although we wrote all symbolic tests, we believe that developers could write them as well, given that the symbolic tests are not syntactically very different from concrete tests. During the symbolic testing of the three libraries, we engaged in discussions with developers who seem to understand the idea behind our symbolic tests. However, in order to make this possible, we would need to improve the debugging facilities of JaVerT.Click first.

Symbolic Testing in Amazon Using JaVerT.Click. In addition to the three open-source libraries `cash`, `p−map` and `webworker-promise`, we also tested the symbolic execution engine of JaVerT.Click in an industrial context inside Amazon. During three months, we collaborated with the Prime Video Automated Reasoning (PVAR) team, whose focus is to develop tools with the use of automatic verification techniques in order to analyse the Prime Video App.[21] We do not provide detailed results as they need to remain confidential. Essentially, we tested three modules of a TypeScript library which

[18] https://github.com/kwolfy/webworker-promise/issues/12
[19] https://github.com/kwolfy/webworker-promise/issues/13
[20] https://github.com/kwolfy/webworker-promise/pull/14
[21] https://www.primevideo.com/splash/t/getTheApp/ref=atv_dl_rdr

is part of the Prime Video App using JaVerT.Click. We chose the targeted modules considering, for instance, their event-driven nature. Before starting the analysis, we ranked the modules in terms of their complexity, taking into consideration for each module: **(1)** the number of lines of code and **(2)** the number of external dependencies. This allowed us to increase the complexity of the analysis over time. Similarly to the approach used for the three open source-libraries, we identified functional properties to be satisfied by the library and developed a symbolic test suite for each targeted module. We found potential issues in two of the three modules tested and reported them to the developers. Additionally, the symbolic tests provided bounded correctness guarantees analogous to the ones obtained for the `cash`, `p—map` and `webworker-promise` libraries.

8. Conclusions

Client-side JavaScript Web programs interact with various event-driven and message-passing APIs. These APIs are inherently complex due to their asynchronous execution model and multi-threaded nature. There are both formal semantics [86, 83, 87, 101, 5] and analysis tools [30, 123, 36, 135, 4, 84, 3] targeting event-driven Web APIs for JavaScript. However, these approaches either support a specific API rather than a variety of APIs or do not faithfully model the targeted APIs. In this thesis, we propose a trusted infrastructure for symbolic analysis of event-driven Web APIs which is built on top of JaVerT 2.0 [35], our state-of-the-art symbolic execution tool for JavaScript which supports three kinds of analysis: *whole-program symbolic testing*, *verification* and *bi-abduction*. We use the symbolic testing mechanism of JaVerT 2.0, which consists of static symbolic execution with boundedness guarantees in the style of Khurshid et al. [76] and Torlak and Bodík [127].

To the best of our knowledge, our infrastructure is the first to support static symbolic execution of event-based Web APIs, the first to explore combinations of such Web APIs, and the first to provide analysis for client-side message-passing Web APIs. We focus on the Web APIs DOM Core Level 1, DOM Events, JS Promises, JS async/await, WebMessaging and WebWorkers, providing trustworthy reference implementations in JavaScript and analysis using JaVerT.Click, which finds real-world bugs and establishes the bounded correctness of functional properties. We summarise our contributions (§8.1) and discuss potential directions for future work (§8.2).

8.1. Summary of Contributions

In the following, we list the novel contributions of this thesis.

Event Semantics. We introduce a general event semantics for event-driven Web programs, identifying event primitives which are enough to capture the event-related behaviour of the Web APIS DOM Events, JS Promises and JS async/await. Our event semantics is designed to be parametric on an underlying language semantics, which can be either concrete or symbolic, thus yielding either a concrete or a symbolic event semantics. In this thesis, we instantiate our event semantics with JSIL, the intermediate goto-language language of JaVerT 2.0. We believe that our event semantics could be instantiated with other languages, such as the intermediate goto-language of the multi-language platform Gillian [32, 91] which evolved from JaVerT 2.0. We also believe that it can be straightforwardly extended to support other event-based APIs, such as File [132] and Timers [137].

Message-passing Semantics. Building on our event semantics, we focus on the message-passing Web APIs, WebMessaging and WebWorkers. These APIs are also event-driven as they rely on DOM Events, so provide further corroboration of our event semantics. They also bring an additional complexity in that they allow JavaScript code to be executed via multiple threads which communicate via messages following the message-passing paradigm [19, 65, 69]. We introduce the message-passing semantics, the first semantics to formalise the message-passing model underneath the WebMessaging and WebWorkers APIs. We design the message-passing semantics parametric on an event semantics, which can be either concrete or symbolic, and a scheduler, which chooses which thread should run at each computation step.

API Reference Implementations. Our event semantics and message-passing semantics do not aim at formalising all features of the targeted APIs; instead, they only capture the core event-driven and message-passing behaviour of JavaScript Web applications, abstracting away details regarding each specific API. These details are modelled by our reference implementations, which interact either with the event semantics or the message-passing semantics when needed, and are trustworthy in that they follow their respective standards line-by-line[1] and pass all applicable tests. For this reason, we believe that our reference implementations could be used in other static analysis tools and serve multiple purposes. For instance, they could support developers on the debugging of JavaScript programs which interact with the APIs.

JaVerT.Click. We implement the event semantics, message-passing semantics and the chosen APIs on top of JaVerT 2.0, obtaining JaVerT.Click, which is the first to implement static symbolic execution of event-based Web APIs, the first to support the combinations of such Web APIs, and the first to provide analysis for WebMessaging and WebWorkers. In JaVerT.Click, we instantiate the event semantics semantics with the JSIL language provided by JaVerT 2.0. Our message-passing is instantiated with our event semantics and a scheduler that runs each thread up to completion, mimicking the behaviour that we observed in most browser implementations.

Symbolic test suites. The symbolic engine of JaVerT.Click has been evaluated against three open source libraries: `cash`, `p−map` and `webworker-promise`. For each chosen library, we developed a symbolic test suite based on existing concrete test suites with the goal of establishing the bounded correctness of several functional properties and find bugs in real-world code. During the symbolic testing process we found, in total, six bugs, four of which have been fixed.

8.2. Future Work

We would like to extend this work in several directions.

Automatic generation of event sequences and thread interleavings. In the context of event-driven programming, there are bugs that are triggered by a particular event sequence. We would like to extend the events module of JaVerT.Click so that it can automatically generate event sequences for the purpose of symbolic testing in the style of [84, 123]. Similarly, the message-passing module of JaVerT.Click could be extended to automatically explore multiple thread interleavings. We believe that the problems of generating event sequences and exploring multiple thread interleavings are analogous and could be tackled with a technique based on Partial Order Reduction (POR) [6], which tries to eliminate paths during symbolic execution that lead to the same outcome, with the goal of avoiding

[1]We do not make this claim for our implementation of JS async/await.

the state explosion problem. This has been applied for event-driven concurrent programs; for instance, Maiya et al. [90] developed a POR-based algorithm for exploring multiple orders of event handlers executions that lead to different states. In order to allow developers to guide the automatic exploration process, we could design a domain specific language for specifying policies for event sequences or thread interleavings. This would reduce the state exploration space even further.

Identifying concurrency bugs. We would like to extend our message-passing module with a mechanism for identifying concurrency bugs, such as *orphan messages* or *bad message interleavings* [128].

A message becomes orphan if it is sent from one thread to another but never handled by the receiving thread. This kind of scenario is typically observed in actor languages such as Erlang and Scala, but can also happen in the context of WebWorkers, for instance, if the main thread sends a message to a worker thread and the worker thread terminates before the message arrives. Although the HTML5 standard [138] mandates that such messages should simply be ignored without raising errors, we could emit warnings so that developers would be aware.

Another scenario that we would like to avoid is the presence of bugs due to the processing of messages in a certain order, corresponding to error triggering message interleavings. The HTML5 standard does not specify in which order messages having different senders should be processed, thus allowing reference implementations to use different policies.

We are not aware of an approach for detecting JavaScript message-passing concurrency bugs, but there are analogous approaches for other languages, such as Erlang [79] and Go [85], which could serve as inspiration.

Instantiating our formal semantics with other languages. Although our event semantics and message-passing semantics are parametric on an underlying language, we have only instantiated them with JSIL. We plan to use our formal semantics in the context of other programming languages, such as Rust.[2] In fact, the Rust language supports message-passing concurrency.[3] We could adapt our message-passing semanticss to deal with the message-passing model of Rust. We hope to transfer the JaVerT.Click infrastructure to Gillian [32, 91], a multi-language platform for symbolic analysis that has evolved directly from JaVerT 2.0. In fact, the JaVerT.Click infrastructure was designed with that transfer in mind. We can then explore our semantics using other programming languages. For example, Sacha-Élie Ayoun is currently working on Gillian Rust, an instantiation of Gillian with sequential Rust. In future, we would like to extend his work to account for message-passing in Rust.

[2] `https://www.rust-lang.org/`.
[3] `https://doc.rust-lang.org/book/ch16-02-message-passing.html`

Bibliography

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93, 2015.

[2] Christoffer Quist Adamsen, Anders Møller, and Frank Tip. Practical Initialization Race Detection for JavaScript Web Applications. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–22, 2017.

[3] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[4] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding Asynchronous Interactions in Full-Stack JavaScript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1169–1180. IEEE, 2016.

[5] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *PACMPL*, 2(OOPSLA):162:1–162:26, 2018.

[6] Rajeev Alur, Robert K Brayton, Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. Partial-order reduction in symbolic state space exploration. In *International Conference on Computer Aided Verification*, pages 340–351. Springer, 1997.

[7] Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Constraint Programming for Dynamic Symbolic Execution of JavaScript. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–19. Springer, 2019.

[8] Roberto Amadini, Graeme Gange, and Peter J Stuckey. Sweep-Based Propagation for String Constraint Solving. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[9] Ellen Arteca, Frank Tip, and Max Schäfer. Enabling Additional Parallelism in Asynchronous JavaScript Applications. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[10] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.

[11] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudžiūnienė, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014.

[12] Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. Toward Automatic Update from Callbacks to Promises. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems*, pages 1–8, 2015.

[13] Achim D Brucker and Michael Herzberg. A Formal Semantics of the Core DOM in Isabelle/HOL. In *Companion Proceedings of the The Web Conference 2018*, pages 741–749, 2018.

[14] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE*, 2011.

[15] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56:82–90, 2013.

[16] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.

[17] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context Logic and Tree Update. In *POPL*, 2005.

[18] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[19] Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development.* " O'Reilly Media, Inc.", 2009.

[20] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. JSExplain: A Double Debugger for JavaScript. In *Companion Proceedings of the The Web Conference 2018*, pages 691–699, 2018.

[21] Maria Christakis and Konstantinos Sagonas. Static Detection of Deadlocks in Erlang. In *Draft Proceedings of the Twelfth International Symposium on Trends in Functional Programming (TFP'11), Department of Computer Systems and Computing, Universidad Complutense de Madrid*, pages 62–76, 2011.

[22] Chromium. Chromium implementation of Web Workers. `https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/core/workers/;l=1?q=workers&ss=chromium%2Fchromium%2Fsrc`, visited 08/2021.

[23] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. Type-aware Concolic Testing of JavaScript Programs. In *Proceedings of the 38th International Conference on Software Engineering*, pages 168–179, 2016.

[24] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Mathew Parkinson, and Hongseok Yang. Views: Compositional Reasoning for Concurrent Programs. In *POPL*, pages 287–300, 2013.

[25] DOM Level 1 Committee. DOM Core Level 1 Tests Repository. `https://dev.w3.org/2001/DOM-Test-Suite/tests/`, visited 08/2021.

[26] ECMA TC39. Test262 Test Suite. `https://github.com/tc39/test262`, visited 05/2020.

[27] ECMA TC39. The ECMAScript Standard - 5th Edition. `https://262.ecma-international.org/5.1/`, visited 08/2021.

[28] ECMA TC39. The ECMAScript Standard - 6th Edition. `http://www.ecma-international.org/ecma-262/6.0/`, visited 08/2021.

[29] ECMA TC39. The ECMAScript Standard - 8th Edition. `https://262.ecma-international.org/8.0/`, visited 08/2021.

[30] Amin Milani Fard, Ali Mesbah, and Eric Wohlstadter. Generating Fixtures for JavaScript Unit Testing (T). In *ASE*, 2015.

[31] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[32] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 927–942, 2020.

[33] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic Execution for JavaScript. In *PPDP*, 2018.

[34] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript Verification Toolchain. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–33, 2017.

[35] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[36] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. Refactoring Asynchrony in JavaScript. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 353–363. IEEE, 2017.

[37] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[38] Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. A Trusted Mechanised Specification of JavaScript: One Year On. In *International Conference on Computer Aided Verification*, pages 3–10. Springer, 2015.

[39] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. Local Hoare Reasoning About DOM. In *PODS*, 2008.

[40] Gargoyle Software Inc. HTMLUnit. `http://htmlunit.sourceforge.io/`, visited 05/2020.

[41] GitHub. cash: Fix #318. https://github.com/fabiospampinato/cash/commit/875944c6568406b3f90d96645eb939aaa294506d, visited 08/2021.

[42] GitHub. cash: Issue #317. https://github.com/fabiospampinato/cash/issues/317, visited 08/2021.

[43] GitHub. cash: Issue #318. https://github.com/fabiospampinato/cash/issues/318, visited 08/2021.

[44] GitHub. p-map: Fix #26. https://github.com/sindresorhus/p-map/commit/b342717a9c44a2d3973b92a7cbf5eaef3a6252a2, visited 08/2021.

[45] GitHub. p-map: Fix #57. https://github.com/sindresorhus/p-map/commit/735d80e928f083c9399e5f741eb1b3cabd0ac1ba, visited 08/2021.

[46] GitHub. p-map: Issue #26. https://github.com/sindresorhus/p-map/issues/26, visited 08/2021.

[47] GitHub. p-map: Issue #57. https://github.com/sindresorhus/p-map/issues/57, visited 08/2021.

[48] GitHub. Web Platform Tests: Issue #29546. https://github.com/web-platform-tests/wpt/issues/29546, visited 08/2021.

[49] GitHub. Web Platform Tests: Issue #29928. https://github.com/web-platform-tests/wpt/issues/29928, visited 08/2021.

[50] GitHub. Web Platform Tests: Issue #29987. https://github.com/web-platform-tests/wpt/issues/29987, visited 08/2021.

[51] GitHub. Web Platform Tests: Issue #31125. https://github.com/web-platform-tests/wpt/issues/31125, visited 08/2021.

[52] GitHub. Web Platform Tests: Issue #33125. https://github.com/web-platform-tests/wpt/issues/33125, visited 08/2021.

[53] GitHub. Web Platform Tests: Pull Request #29546. https://github.com/web-platform-tests/wpt/pull/29546, visited 08/2021.

[54] GitHub. Web Platform Tests: Pull Request #29987. https://github.com/web-platform-tests/wpt/pull/29987, visited 08/2021.

[55] GitHub. Web Platform Tests: Pull Request #29988. https://github.com/web-platform-tests/wpt/pull/29988, visited 08/2021.

[56] GitHub. Web Platform Tests: Pull Request #31673. https://github.com/web-platform-tests/wpt/pull/31673, visited 08/2021.

[57] GitHub. Web Platform Tests Repository. https://github.com/web-platform-tests/wpt/, visited 08/2021.

[58] GitHub. webworker-promise: Issue #13. `https://github.com/kwolfy/webworker-promise/issues/13`, visited 08/2021.

[59] GitHub. webworker-promise: Issue #9. `https://github.com/kwolfy/webworker-promise/issues/9`, visited 08/2021.

[60] GitHub. webworker-promise: Issue #9. `https://github.com/kwolfy/webworker-promise/issues/12`, visited 08/2021.

[61] GitHub. webworker-promise: Pull Request #11. `https://github.com/kwolfy/webworker-promise/pull/11`, visited 08/2021.

[62] GitHub. webworker-promise: Pull Request #14. `https://github.com/kwolfy/webworker-promise/pull/14`, visited 08/2021.

[63] Patrice Godefroid. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 1–1, 2007.

[64] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[65] Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft: An Exploratory Survey. In *CAV workshop on exploiting concurrency efficiently and correctly*. Princeton, USA, 2008.

[66] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. Automatic Migration From Synchronous to Asynchronous JavaScript APIs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[67] Ido Green. *Web workers: Multithreaded Programs in JavaScript.* " O'Reilly Media, Inc.", 2012.

[68] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *European conference on Object-oriented programming*, pages 126–150. Springer, 2010.

[69] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[70] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15, 2014.

[71] Casper S Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless Model Checking of Event-Driven Applications. *ACM SIGPLAN Notices*, 50(10):57–73, 2015.

[72] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 59–69, 2011.

[73] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *ESEC/FSE*, 2011.

[74] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.

[75] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, pages 320–339. Springer, 2010.

[76] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[77] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-Force: Forced Execution on JavaScript. In *Proceedings of the 26th international conference on World Wide Web*, pages 897–906. International World Wide Web Conferences Steering Committee, 2017.

[78] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. *Web Application Security Consortium, Articles*, 4:365–372, 2005.

[79] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In *International Symposium on Functional and Logic Programming*, pages 247–263. Springer, 2018.

[80] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and Safety for Channel-Based Programming. *ACM SIGPLAN Notices*, 52(1):748–761, 2017.

[81] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, volume 10. Citeseer, 2012.

[82] Sebastian Lekies, Ben Stock, and Martin Johns. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204, 2013.

[83] Benjamin S. Lerner, Matthew J. Carroll, Dan P. Kimmel, Hannah Quay-De La Vallee, and Shriram Krishnamurthi. Modeling and Reasoning About DOM Events. In *WebApps*, 2012.

[84] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459, 2014.

[85] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 616–629, 2021.

[86] Matthew C Loring, Mark Marron, and Daan Leijen. Semantics of Asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, pages 51–62, 2017.

[87] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A Model for Reasoning about JavaScript Promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017.

[88] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node. js javascript applications. *ACM SIGPLAN Notices*, 50(10):505–519, 2015.

[89] Sergio Maffeis, John C Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, pages 307–325. Springer, 2008.

[90] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-Threaded Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 680–697. Springer, 2016.

[91] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In *International Conference on Computer Aided Verification*, pages 827–850. Springer, 2021.

[92] Ana Almeida Matos, José Fragoso Santos, and Tamara Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *TGC*, 2014.

[93] Mozilla Web Docs. Same-origin policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`, visited 08/2021.

[94] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64. IEEE, 2013.

[95] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. DexterJS: Robust Testing Platform for DOM-Based XSS Vsulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 946–949, 2015.

[96] Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *ASE*, 2015.

[97] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356, 2015.

[98] Joonyoung Park, Kwangwon Sun, and Sukyoung Ryu. EventHandler-Based Analysis Framework for Web Apps Using Dynamically Collected States. In *International Conference on Fundamental Approaches to Software Engineering*, pages 129–145. Springer, 2018.

[99] Joe Gibbs Politz, Matthew J Carroll, Benjamin S Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 1–16, 2012.

[100] Azalea Raad, José Fragoso Santos, and Philippa Gardner. DOM: Specification and Client Reasoning. In *Asian Symposium Programming Languages and Systems, APLAS*, pages 401–422, 2016.

[101] Vineet Rajani, Abhishek Bichhawat, Deepak Garg, and Christian Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015.

[102] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 151–166, 2013.

[103] John C Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[104] Grigore Roșu and Traian Florin Șerbănuță. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[105] Ruslan Boliev. Webworker-promise library. `https://github.com/kwolfy/webworker-promise`, visited 08/2021.

[106] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *ESORICS*, 2009.

[107] Gabriela Sampaio, José Fragoso Santos, Petar Maksimović, and Philippa Gardner. A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[108] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript Verification Toolchain. *PACMPL*, 2(POPL):50:1–50:33, 2018.

[109] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[110] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDss*, 2010.

[111] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[112] Koushik Sen. DART: Directed Automated Random Testing. In *Haifa Verification Conference*, volume 6405, page 4, 2009.

[113] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[114] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[115] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE'13*, 2013.

[116] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853, 2015.

[117] Mukesh Singhal and Ajay Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[118] Gareth Smith. *Local Reasoning About Web Programs*. PhD thesis, Imperial College, UK, 2011.

[119] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS*, 2013.

[120] S. Sorhus. p-map (GitHub). `https://github.com/sindresorhus/p-map`, visited 05/2020.

[121] Marius Steffens and Ben Stock. PMForce: Systematically Analyzing postMessage Handlers at Scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 493–505, 2020.

[122] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient Dynamic Analysis for Node. js. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 196–206, 2018.

[123] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static DOM Event Dependency Analysis for Testing Web Applications. In *FSE*, 2016.

[124] Peter Thiemann. A Type Safe DOM API. In *DBPL*, 2005.

[125] Emina Torlak and Rastislav Bodík. Growing Solver-Aided Languages with Rosette. In *Onward!*, 2013.

[126] Emina Torlak and Rastislav Bodík. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*, 2014.

[127] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. *ACM SIGPLAN Notices*, 49(6):530–541, 2014.

[128] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A Study of Concurrency Bugs and Advanced Development Support for Actor-Based Programs. In *Programming with Actors*, pages 155–185. Springer, 2018.

[129] W3C. DOM Core Level 1 Official Test Suite. `http://www.w3.org/2004/04/ecmascript/level1/core/`, visited 05/2020.

[130] W3C. DOM Core Level 1 Specification. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html, visited 05/2020.

[131] W3C. DOM Core Level 3 Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/, visited 05/2020.

[132] W3C. File API. http://www.w3.org/TR/FileAPI/, visited 05/2020.

[133] W3C. HTML Standard. http://html.spec.whatwg.org/multipage/workers.html#workers, visited 05/2020.

[134] W3C. W3C Community. https://www.w3.org/, visited 08/2021.

[135] Jie Wang, Wensheng Dou, Chushu Gao, Yu Gao, and Jun Wei. Context-Based Event Trace Reduction in Client-Side JavaScript Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 127–138. IEEE, 2018.

[136] WHATWG. DOM API Specification. http://dom.spec.whatwg.org, visited 05/2020.

[137] WHATWG. Timers API. https://html.spec.whatwg.org/multipage/timers-and-user-prompts.html#timers, visited 05/2020.

[138] WHATWG. HTML5 Standard. https://html.spec.whatwg.org/multipage/, visited 08/2021.

[139] WHATWG. The DOM API (Events). https://html.spec.whatwg.org/multipage/webappapis.html#event-handler-attributes, visited 08/2021.

[140] WHATWG. The postMessage API. https://html.spec.whatwg.org/multipage/web-messaging.html#posting-messages, visited 08/2021.

[141] WHATWG. WHATWG Community. https://whatwg.org/, visited 08/2021.

[142] Ken Wheeler, Shaw, and Fabio Spampinato. cash (GitHub). https://github.com/kenwheeler/cash, visited 05/2020.

[143] Allen Wirfs-Brock and Brendan Eich. JavaScript: The First 20 Years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–189, 2020.

[144] Adam Wright. *Structural Separation Logic*. PhD thesis, Imperial College London, UK, 2013.

[145] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

[146] Yousuf Nashwaan. xml-js: Converter Utility Between XML Text and JavaScript Objects/JSON Text. http://www.npmjs.com/package/xml-js, visited 05/2020.

[147] Yunxiao Zou, Zhenyu Chen, Yunhui Zheng, Xiangyu Zhang, and Zebao Gao. Virtual DOM Coverage: Drive an Effective Testing for Dynamic Web Applications. *ISSTA*, 2014.

A. E-semantics

A.1. Concrete E-semantics

Values: $v \in \mathcal{V}$ **Event Types:** $e \in \mathcal{E} \subset \mathcal{V}$
Function Identifiers: $f \in \mathcal{F} \subset \mathcal{V}$ **Language Configurations:** $lc \in \mathcal{LC}$
Configuration Predicates: $\rho \in \mathcal{P} : \mathcal{LC} \to \mathbb{B}$ **Handler Registers:** $h \in \mathcal{H} : \mathcal{E} \rightharpoonup \overline{\mathcal{F}}$
Continuations: $\kappa \in \mathcal{K} := (f, v) \mid (lc, \rho)$ **Continuation Queues:** $q \in \mathcal{Q} : \overline{\mathcal{K}}$
Event Configurations: $\epsilon c \in \mathcal{EC} : \mathcal{LC} \times \mathcal{H} \times \mathcal{Q}$
Event Primitives: $\mathrm{p} \in \mathrm{P} := \cdot \mid \mathsf{addHdlr}\langle e, f\rangle \mid \mathsf{remHdlr}\langle e, f\rangle \mid \mathsf{sDispatch}\langle e, v\rangle \mid \mathsf{aDispatch}\langle e, vs\rangle \mid \mathsf{schedule}\langle f, vs\rangle \mid \mathsf{await}\langle v, \rho\rangle$

Figure A.1.: Events Syntax

Language Semantics Interface

1. $\mathsf{initialConf}(lc, (f, e, v)) = lc'$, where lc' has the heap component of lc and the control flow/store components required for starting the execution of the handler f with arguments e and v
2. $\mathsf{mergeConfs}(lc, lc') = lc''$, where lc'' has the heap component of lc and the control flow/store components of lc'
3. $\mathsf{isFinal}(lc) = \texttt{true}$, if lc is final; and $= \texttt{false}$ otherwise
4. $\mathsf{interrupt}(lc) = lc'$, where lc' is the same as lc, except that it is marked as final
5. $\mathsf{splitReturn}(lc, v) = (lc_r, lc_a)$, where lc_r is obtained from lc by setting up the control flow component as if the currently executing function, f, returned the value v and lc_a is obtained from lc by setting up the control flow component to only contain the remainder of the execution of f

$\mathcal{AH} : \mathcal{H} \times \mathcal{E} \times \mathcal{F} \to \mathcal{H}$

$\mathcal{AH}(h, e, f)$ adds the handler f for the event e to the handler register h.

Add Handler

$$\mathcal{AH}(h, e, f) \triangleq \begin{cases} h\,[e \mapsto h(e) +\!\!+ [f]\,], & \text{if } e \in \mathtt{dom}(h) \\ h\,[e \mapsto [f]\,], & \text{otherwise} \end{cases}$$

$\mathcal{RH} : \mathcal{H} \times \mathcal{E} \times \mathcal{F} \to \mathcal{H}$

$\mathcal{RH}(h, e, f)$ removes the handler f for the event e from the handler register h.

Remove Handler

$$\mathcal{RH}(h, e, f) \triangleq \begin{cases} h\,[e \mapsto h(e) \setminus f\,], & \text{if } e \in \mathtt{dom}(h) \\ h, & \text{otherwise} \end{cases}$$

$\mathcal{FH} : \mathcal{H} \times \mathcal{E} \to \mathcal{H}$

$\mathcal{FH}(h, e)$ finds all of the handlers for the event e in the handler register h.

Find Handlers

$$\mathcal{FH}(h, e) \triangleq \begin{cases} h(e), & \text{if } e \in \mathtt{dom}(h) \\ [\,], & \text{otherwise} \end{cases}$$

$\mathcal{CW}_{\mathrm{L}} : \mathcal{LC} \times \mathcal{K} \to \mathcal{LC}$

$\mathcal{CW}_{\mathrm{L}}(lc, \kappa)$ sets up the configuration for the execution of the continuation κ, starting from the configuration lc.

Continue With - Handler-Continuation

$$\mathcal{CW}_{\mathrm{L}}(lc, (f, e, v)) \triangleq \mathrm{L}.\mathsf{initialConf}(lc, (f, e, v))$$

Continue With - Yield-Continuation

$$\frac{\rho(lc) = \mathsf{True}}{\mathcal{CW}_{\mathrm{L}}(lc, (lc', \rho)) \triangleq \mathrm{L}.\mathsf{mergeConfs}(lc, lc')}$$

Auxiliary Functions of the E-semantics

Language Transition

$$\frac{lc \leadsto_{\mathrm{L}} lc'}{\langle lc, h, q \rangle \leadsto_{\mathsf{E}} \langle lc', h, q \rangle}$$

No action is performed—the handler register and the continuation queue remain unchanged

Add Handler

$$\frac{lc \leadsto^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{addHdlr}\langle e, f \rangle}{\langle lc, h, q \rangle \leadsto_{\mathsf{E}} \langle lc', \mathcal{AH}(h, e, f), q \rangle}$$

The new handler for the given event is registered in the handler register

Remove Handler

$$\frac{c \leadsto^{\mathrm{p}}_{\mathrm{L}} c' \quad \mathrm{p} = \mathsf{remHdlr}\langle e, f \rangle}{\langle lc, h, q \rangle \leadsto_{\mathsf{E}} \langle lc', \mathcal{RH}(h, e, f), q \rangle}$$

The given handler is de-registered for the given event from the handler register

Synchronous Dispatch

$$\frac{\begin{array}{c} lc \leadsto^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{sDispatch}\langle e, vs \rangle \\ [f_i \mid_0^n] = \mathcal{FH}(h, e) \quad q' = [(f_i, vs) \mid_{i=0}^n] \\ lc'' = \mathrm{L}.\mathsf{suspend}(lc') \end{array}}{\langle lc, h, q \rangle \leadsto_{\mathsf{E}} \langle lc'', h, q' +\!\!+ [(lc', (\lambda lc.\mathsf{True}))] +\!\!+ q \rangle}$$

The execution of the current UL-configuration lc' is interrupted; a list of handler-continuations based on the handlers registered for the dispatched event is placed at the *front* of the continuation queue, together with a yield-continuation that unconditionally resumes the execution of lc'

Asynchronous Dispatch

$$\frac{\begin{array}{c} lc \leadsto^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{aDispatch}\langle e, vs \rangle \\ [f_i \mid_0^n] = \mathcal{FH}(h, e) \quad q' = [(f_i, vs) \mid_{i=0}^n] \end{array}}{\langle lc, h, q \rangle \leadsto_{\mathsf{E}} \langle lc', h', q +\!\!+ q' \rangle}$$

A list of handler-continuations based on the handlers registered for the dispatched event is placed at the *back* of the continuation queue

Schedule

$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{schedule} f, vs \qquad q' = q' +\!\!+ [(f, vs)]}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc, h, q'\rangle}$$

The execution of the function f with parameters $\bar{v}$ is placed at the *back* of the continuation queue

Await

$$\frac{lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \quad \mathrm{p} = \mathsf{await}\langle v, \rho\rangle \qquad (lc_r, lc_a) = \mathrm{L}.\mathsf{splitReturn}(lc', v)}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc_r, h, q +\!\!+ [(lc_a, \rho)]\rangle}$$

The execution proceeds with the return configuration; the await configuration is placed at the *back* of the continuation queue

Environment Dispatch

$$\frac{[f_i \mid_0^n] = \mathcal{FH}(h, e) \qquad q' = [(f_i, vs) \mid_{i=0}^n]}{\langle lc, h, q\rangle \rightsquigarrow^{\mathsf{fire}\langle e, vs\rangle}_{\mathsf{E}} \langle lc, h, q +\!\!+ q'\rangle}$$

An environment-dispatched event is essentially treated in the same way as an asynchronously-dispatched event.

Continuation - Success

$$\frac{\mathrm{L}.\mathsf{final}(lc) \quad q = \kappa : q'}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle \mathcal{CW}_{\mathrm{L}}(lc, \kappa), h, q'\rangle}$$

The current UL-configuration is final; the configuration at the front of the continuation queue is set up for execution

Continuation - Failure

$$\frac{\mathrm{L}.\mathsf{final}(lc) \quad q = \kappa : q' \qquad (lc, \kappa) \notin \mathtt{dom}(\mathcal{CW}_{\mathrm{L}})}{\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc, h, q' +\!\!+ [\kappa]\rangle}$$

The current UL-configuration is final; the configuration at the front of the continuation queue cannot be executed and is placed at the *back* of the continuation queue

Transition System: $\langle lc, h, q\rangle \rightsquigarrow^{\epsilon\alpha}_{\mathsf{E}} \langle lc', h', q'\rangle$

A.2. Symbolic E-semantics

Values: $\hat{v} \in \widehat{\mathcal{V}}$ **Event Types:** $\hat{e} \in \widehat{\mathcal{E}} \subset \widehat{\mathcal{V}}$

Function Identifiers: $f \in \mathcal{F} \subset \widehat{\mathcal{V}}$ **Path Conditions:** $\pi \in \Pi \subset \widehat{\mathcal{V}}$

Language Configurations: $\widehat{lc} \in \widehat{\mathcal{LC}}$ **Configuration Predicates:** $\hat{\rho} \in \widehat{\mathcal{P}} : \widehat{\mathcal{LC}} \to \mathbb{B}$

Handler Registers: $\hat{h} \in \widehat{\mathcal{H}} : \widehat{\mathcal{E}} \rightharpoonup \overline{\mathcal{F}}$ **Continuations:** $\hat{\kappa} \in \widehat{\mathcal{K}} := (f, \hat{v}) \mid (\widehat{lc}, \hat{\rho})$

Continuation Queues: $\hat{q} \in \widehat{\mathcal{Q}} : \overline{\widehat{\mathcal{K}}}$ **Event Configurations:** $\widehat{ec} \in \widehat{\mathcal{EC}} : \widehat{\mathcal{LC}} \times \widehat{\mathcal{H}} \times \widehat{\mathcal{Q}}$

Event Primitives: $\hat{\mathrm{p}} \in \widehat{\mathrm{P}} := \mathsf{addHdlr}\langle \hat{e}, f\rangle \mid \mathsf{remHdlr}\langle \hat{e}, f\rangle \mid \mathsf{sDispatch}\langle \hat{e}, \hat{v}\rangle \mid \mathsf{aDispatch}\langle \hat{e}, \hat{v}\rangle$
$\mid \mathsf{schedule} f, \bar{\hat{v}} \mid \mathsf{await}\langle \hat{v}, \hat{\rho}\rangle \mid \cdot$

Figure A.2.: Events Syntax

Auxiliary Functions of the Language Semantics

1. $\mathsf{initialConf}(\widehat{lc}, (f, \hat{e}, \hat{v})) = \widehat{lc}'$, where $\widehat{lc}'$ has the heap component of $\widehat{lc}$ and the control flow/store components required for starting the execution of the handler f with arguments $\hat{e}$ and $\hat{v}$

2. $\mathsf{mergeConfs}(\widehat{lc}, \widehat{lc}') = \widehat{lc}''$, where $\widehat{lc}''$ has the heap component of $\widehat{lc}$ and the control flow/store components of $\widehat{lc}'$

3. $\mathsf{isFinal}(\widehat{lc}) = \texttt{true}$, if $\widehat{lc}$ is final; and $= \texttt{false}$ otherwise.

4. $\mathsf{interrupt}(\widehat{lc}) = \widehat{lc}'$, where $\widehat{lc}'$ is the same as $\widehat{lc}$, except that it is marked as final

5. $\mathsf{assume}(\widehat{lc}, \pi) = \widehat{lc}'$, where $\widehat{lc}'$ is obtained from $\widehat{lc}$ by extending its path condition with the formula π, if such an extension is satisfiable

6. $\mathsf{splitReturn}(\widehat{lc}, \hat{v}) = (\widehat{lc}_r, \widehat{lc}_a)$, where $\widehat{lc}_r$ is obtained from lc by setting up the control flow component as if the currently executing function, f, returned the value $\hat{v}$ and $\widehat{lc}_a$ is obtained from $\widehat{lc}$ by setting up the control flow component to only contain the remainder of the execution of f

7. $\mathsf{pc}(\widehat{lc}) = \pi$, where π is the path condition computed in the current branch of configuration $\widehat{lc}$. We leave the computation of the path condition to the underlying language L, meaning that $\mathsf{pc}(\langle \widehat{lc}, \hat{h}, \hat{q} \rangle) = \mathsf{pc}(\widehat{lc})$

$\mathcal{AH} : \widehat{\mathcal{H}} \times \widehat{\mathcal{E}} \times \mathcal{F} \to \wp(\widehat{\mathcal{H}} \times \Pi)$

$\mathcal{AH}(\hat{h}, \hat{e}, f)$ adds the handler f for the event $\hat{e}$ to the handler register $\hat{h}$, branching on all possible values of $\hat{e}$.

Add Handler

$$\mathcal{AH}(\hat{h}, \hat{e}, f) \triangleq \begin{array}{l} \left\{ \, (\hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') +\!\!+ [f]\right], \hat{e} = \hat{e}') \mid \hat{e}' \in \mathtt{dom}(\hat{h}) \, \right\} \\ \cup \left\{ \, (h\left[e \mapsto [f]\right], \hat{e}' \in \mathtt{dom}(\hat{h})) \, \right\} \end{array}$$

$\mathcal{RH} : \widehat{\mathcal{H}} \times \widehat{\mathcal{E}} \times \mathcal{F} \to \wp(\widehat{\mathcal{H}} \times \Pi)$

$\mathcal{RH}(h, e, f)$ removes the handler f for the event $\hat{e}$ from the handler register $\hat{h}$, branching on all possible values of $\hat{e}$.

Remove Handler

$$\mathcal{RH}(\hat{h}, \hat{e}, f) \triangleq \begin{array}{l} \left\{ \, (\hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') \setminus f\right], \hat{e} = \hat{e}') \mid \hat{e}' \in \mathtt{dom}(\hat{h}) \, \right\} \\ \cup \left\{ \, (h, \hat{e}' \in \mathtt{dom}(\hat{h})) \, \right\} \end{array}$$

$\mathcal{FH} : \widehat{\mathcal{H}} \times \widehat{\mathcal{E}} \to \wp(\widehat{\mathcal{H}} \times \Pi)$

$\mathcal{FH}(\hat{h}, \hat{e})$ finds all of the handlers for the event $\hat{e}$ in the handler register $\hat{h}$, branching on all possible values of $\hat{e}$.

Find Handlers

$$\mathcal{FH}(\hat{h}, \hat{e}) \triangleq \begin{array}{l} \left\{ \, (\hat{h}(\hat{e}), \hat{e} = \hat{e}') \mid \hat{e}' \in \mathtt{dom}(\hat{h}) \, \right\} \\ \cup \left\{ \, ([\,], \hat{e}' \in \mathtt{dom}(\hat{h})) \, \right\} \end{array}$$

$\mathcal{CW}_{\mathrm{L}} : \widehat{\mathcal{LC}} \times \widehat{\mathcal{K}} \to \widehat{\mathcal{LC}}$

$\mathcal{CW}_{\mathrm{L}}(\widehat{lc}, \hat{\kappa})$ sets up the configuration for the execution of the continuation $\hat{\kappa}$, starting from the configuration $\widehat{lc}$.

Continue With - Handler-Continuation

$$\mathcal{CW}_{\mathrm{L}}(\widehat{lc}, (f, \hat{v})) \triangleq \mathsf{L.initialConf}(\widehat{lc}, (f, \hat{v}))$$

Continue With - Yield-Continuation

$$\frac{\hat{\rho}(\widehat{lc}) = \mathsf{True}}{\mathcal{CW}_{\mathrm{L}}(\widehat{lc}, (\widehat{lc}', \hat{\rho})) \triangleq \mathsf{L.mergeConfs}(\widehat{lc}, \widehat{lc}')}$$

Auxiliary Relations of the E-semantics (In Set-Notation)

Transition System: $\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow^{\hat{\epsilon\alpha}}_{\hat{\mathsf{E}}} \langle \widehat{lc}', \hat{h}', \hat{q}'\rangle$ The transition rules for the symbolic E-semantics differ from the concrete ones in that every time an auxiliary relation is used, the constraint it generates must be added to the current path condition using the assume function of the UL-semantics. These differences are highlighted in grey.

Language Transition

$$\frac{\widehat{lc} \rightsquigarrow_{\mathrm{L}} \widehat{lc}'}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}} \langle \widehat{lc}', \hat{h}, \hat{q}\rangle}$$

Add Handler

$$\frac{\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{addHdlr}\langle \hat{e}, f\rangle \quad \mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi) \quad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}} \langle \widehat{lc}'', \hat{h}', \hat{q}\rangle}$$

Remove Handler

$$\frac{\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{remHdlr}\langle \hat{e}, f\rangle \quad \mathcal{RH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi) \quad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}} \langle \widehat{lc}'', \hat{h}', \hat{q}\rangle}$$

Synchronous Dispatch

$$\frac{\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{sDispatch}\langle \hat{e}, \hat{vs}\rangle \quad \mathcal{FH}((\hat{h}, \hat{e})) \rightsquigarrow (([f_i \mid_0^n], \pi)) \quad \hat{q}' = [(f_i, \hat{vs}) \mid_{i=0}^n] \quad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi) \quad lc''' = \mathrm{L}.\mathsf{suspend}(lc'')}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}} \langle \widehat{lc}''', \hat{h}, \hat{q}' +\!\!+ [(\widehat{lc}'', (\lambda \widehat{lc}.\mathsf{True}))] +\!\!+ \hat{q}\rangle}$$

Asynchronous Dispatch

$$\frac{\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{aDispatch}\langle \hat{e}, \hat{vs}\rangle \quad \mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, \hat{v}) \mid_{i=0}^n] \quad \widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}} \langle \widehat{lc}'', \hat{h}', \hat{q}'\rangle}$$

Schedule

$$\frac{\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{schedule} f, \hat{vs} \quad \hat{q}' = \hat{q}' +\!\!+ [(f, \hat{vs})]}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\mathsf{E}} \langle \widehat{lc}, \hat{h}, \hat{q}'\rangle}$$

Await

$$\frac{\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \quad \hat{\mathrm{p}} = \mathsf{await}\langle \hat{v}, \hat{\rho}\rangle \quad (\widehat{lc}_r, \widehat{lc}_a) = \mathrm{L}.\mathsf{splitReturn}(\widehat{lc}', \hat{v})}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\mathsf{E}} \langle \widehat{lc}_r, \hat{h}, \hat{q} +\!\!+ [(\widehat{lc}_a, \hat{\rho})]\rangle}$$

Environment Dispatch

$$\frac{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, \hat{e}, \hat{v}) \mid_{i=0}^n] \quad \widehat{lc}' = \mathrm{L}.\mathsf{assume}(\widehat{lc}, \pi)}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow^{(\hat{e}, \hat{v})}_{\hat{\mathsf{E}}} \langle \widehat{lc}', \hat{h}, \hat{q} +\!\!+ \hat{q}'\rangle}$$

Continuation - Success

$$\frac{\mathrm{L}.\mathsf{final}(\widehat{lc}) \quad \hat{q} = \hat{\kappa} : \hat{q}'}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\mathsf{E}} \langle \mathcal{CW}_{\mathrm{L}}(\widehat{lc}, \hat{\kappa}), \hat{h}, \hat{q}'\rangle}$$

Continuation - Failure

$$\frac{\mathrm{L}.\mathsf{final}(\widehat{lc}) \quad \hat{q} = \hat{\kappa} : \hat{q}' \quad (\widehat{lc}, \hat{\kappa}) \notin \mathtt{dom}(\mathcal{CW}_{\mathrm{L}})}{\langle \widehat{lc}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\mathsf{E}} \langle \widehat{lc}, \hat{h}, \hat{q}' +\!\!+ [\hat{\kappa}]\rangle}$$

A.3. Correctness

Type-Preserving Symbolic Environments: $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$

Interpretations ($\mathcal{I}_\varepsilon(\hat{v})$): $\mathcal{I}_\varepsilon(\hat{x}) = \varepsilon(\hat{x})$

Interpretation of E-semantics Structures

HR - Empty
$$\mathcal{I}_\varepsilon(\emptyset) \triangleq \emptyset$$

HR - Composition
$$\mathcal{I}_\varepsilon(\hat{h}_1 \uplus \hat{h}_2) \triangleq \mathcal{I}_\varepsilon(\hat{h}_1) \uplus \mathcal{I}_\varepsilon(\hat{h}_2)$$

HR - Cell
$$\mathcal{I}_\varepsilon([\hat{e} \mapsto \overline{f}]) \triangleq [\mathcal{I}_\varepsilon(\hat{e}) \mapsto \overline{f}]$$

CQ - Empty
$$\mathcal{I}_\varepsilon([]) \triangleq []$$

CQ - Non-Empty
$$\mathcal{I}_\varepsilon(\hat{\kappa} : \hat{q}) \triangleq \mathcal{I}_\varepsilon(\hat{\kappa}) : \mathcal{I}_\varepsilon(\hat{q})$$

Cont - Hanlder-Cont
$$\mathcal{I}_\varepsilon((f, [\hat{v}_1, ..., \hat{v}_n])) \triangleq (f, [\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n)])$$

Cont - Yield-Cont
$$\mathcal{I}_\varepsilon((\widehat{lc}, \hat{\rho})) \triangleq (\mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{\rho}))$$

Event Label - AH/RH
$$\frac{\mathsf{lab} \in \{\mathsf{addHdlr}, \mathsf{remHdlr}\}}{\mathcal{I}_\varepsilon(\mathsf{lab}\langle \hat{e}, f\rangle) \triangleq \mathsf{lab}\langle \mathcal{I}_\varepsilon(\hat{e}), f\rangle}$$

Event Label - SD/AD
$$\frac{\mathsf{lab} \in \{\mathsf{sDispatch}, \mathsf{aDispatch}\}}{\mathcal{I}_\varepsilon(\mathsf{lab}\langle \hat{e}, \hat{v}\rangle) \triangleq \mathsf{lab}\langle \mathcal{I}_\varepsilon(\hat{e}), \mathcal{I}_\varepsilon(\hat{v})\rangle}$$

Event Label - Schedule
$$\mathcal{I}_\varepsilon(\mathsf{schedule} f, [\hat{v}_1, ..., \hat{v}_n]) \triangleq \mathsf{schedule} f, [\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n)]$$

Event Label - Await
$$\mathcal{I}_\varepsilon(\mathsf{await}\langle \hat{\rho}\rangle) \triangleq \mathsf{await}\langle \mathcal{I}_\varepsilon(\hat{\rho})\rangle$$

EAction - Event
$$\mathcal{I}_\varepsilon((\hat{e}, \hat{v})) \triangleq (\mathcal{I}_\varepsilon(\hat{e}), \mathcal{I}_\varepsilon(\hat{v}))$$

EAction - UL
$$\mathcal{I}_\varepsilon(\cdot) \triangleq \cdot$$

ESemantics Configuration
$$\mathcal{I}_\varepsilon(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle) \triangleq \langle \mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q})\rangle$$

Models of Symbolic E-semantics Structures

L-Configurations
$$\mathcal{M}_\pi(\widehat{lc}) \triangleq \{(\varepsilon, \mathcal{I}_\varepsilon(\widehat{lc})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$$

E-Configurations
$$\mathcal{M}_\pi(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle) \triangleq \{(\varepsilon, \langle \mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q}))\rangle \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$$

Event Labels
$$\mathcal{M}_\pi(\hat{\mathrm{p}}) \triangleq \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\mathrm{p}})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$$

Environment Actions
$$\mathcal{M}_\pi(\hat{\epsilon\alpha}) \triangleq \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\epsilon\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$$

Requirements 1 (Interpretations)**.** *Interpretations on symbolic values must satisfy the following properties:*

1. $\mathcal{I}_\varepsilon(v) = v$

2. $\mathcal{I}_\varepsilon(l + l') = \mathcal{I}_\varepsilon(l) + \mathcal{I}_\varepsilon(l')$

3. $\hat{\rho}(\widehat{lc}) \iff (\mathcal{I}_\varepsilon(\hat{\rho}))(\mathcal{I}_\varepsilon(\widehat{lc}))$

Requirements 2 (L-Semantics Interface Functions)**.** *The interface functions of the L-semantics must preserve path conditions, as follows:*

1. $\mathsf{assume}(\widehat{lc}, \pi) = \widehat{lc}' \implies \mathsf{pc}(\widehat{lc}') = \mathsf{pc}(\widehat{lc}) \wedge \pi$

2. $\mathsf{pc}(\mathsf{initialConf}(\widehat{lc}, (f, \hat{v})) = \mathsf{pc}(\widehat{lc})$

3. $\mathsf{pc}(\mathsf{mergeConfs}(\widehat{lc}, \widehat{lc}')) = \mathsf{pc}(\widehat{lc})$

4. $\mathsf{splitReturn}(\widehat{lc}, \hat{v}) = (\widehat{lc}_r, \widehat{lc}_a) \implies \mathsf{pc}(\widehat{lc}) = \mathsf{pc}(\widehat{lc}_r) \wedge \mathsf{pc}(\widehat{lc}) = \mathsf{pc}(\widehat{lc}_a)$

5. $\mathsf{pc}(\mathsf{interrupt}(\widehat{lc})) \implies \mathsf{pc}(\widehat{lc})$

Requirements 3 (Interpretation Preservation)**.** *The interface functions of the L-semantics must preserve interpretations, as follows:*

1. $\mathsf{initialConf}(\widehat{lc}, (f, \hat{v})) = \widehat{lc}' \implies \mathsf{initialConf}(\mathcal{I}_\varepsilon(\widehat{lc}), (f, \mathcal{I}_\varepsilon(\hat{v}))) = \mathcal{I}_\varepsilon(\widehat{lc}')$
2. $\mathsf{mergeConfs}(\widehat{lc}, \widehat{lc}') = \widehat{lc}'' \implies \mathsf{mergeConfs}(\mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\widehat{lc}')) = \mathcal{I}_\varepsilon(\widehat{lc}'')$
3. $\mathsf{isFinal}(\widehat{lc}) \iff \mathsf{isFinal}(\mathcal{I}_\varepsilon(\widehat{lc}))$
4. $\mathsf{assume}(\widehat{lc}, \pi_a) = \widehat{lc}' \wedge \mathcal{I}_\varepsilon(\mathsf{pc}(\widehat{lc}')) = \mathsf{True} \implies \mathcal{I}_\varepsilon(\widehat{lc}) = \mathcal{I}_\varepsilon(\widehat{lc}')$
5. $\mathsf{interrupt}(\widehat{lc}) = \widehat{lc}' \implies \mathsf{interrupt}(\mathcal{I}_\varepsilon(\widehat{lc})) = \mathcal{I}_\varepsilon(\widehat{lc}')$
6. $\mathsf{splitReturn}(\widehat{lc}, \hat{v}) = (\widehat{lc}_r, \widehat{lc}_a) \implies \mathsf{splitReturn}(\mathcal{I}_\varepsilon(\widehat{lc}), \mathcal{I}_\varepsilon(\hat{v})) = (\mathcal{I}_\varepsilon(\widehat{lc}_r), \mathcal{I}_\varepsilon(\widehat{lc}_a))$

Lemma 4 (Configuration Projection - Models)**.**

$$(\varepsilon, \langle lc, h, q\rangle) \in \mathcal{M}_\pi(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle) \implies (\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$$

PROOF:
ASSUME: 1. $(\varepsilon, \langle lc, h, q\rangle) \in \mathcal{M}_\pi(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle)$
PROVE: $lc \in \mathcal{M}_\pi(\widehat{lc})$

1. $(\varepsilon, \langle lc, h, q\rangle) \in \mathcal{M}_\pi(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle)$ [Assumption 1]
2. $(\varepsilon, \langle lc, h, q\rangle) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\langle lc, h, q\rangle)) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [Definition of $\mathcal{M}_\pi()$]
3. $\langle lc, h, q\rangle = \mathcal{I}_\varepsilon(\langle \widehat{lc}, \hat{h}, \hat{q}\rangle)$ [Set theory and 2]
4. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [Definition of $\mathcal{M}_\pi()$]
5. $lc = \mathcal{I}_\varepsilon(\widehat{lc})$ [Equality of Tuples and 4]
6. $(\varepsilon, lc) \in \{(\varepsilon, \widehat{lc}) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [Set theory and 5]
7. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [Definition of $\mathcal{M}_\pi()$ and 6]

□

Lemma 5 (Add Handler - Symbolic to Concrete)**.**

$$\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi) \wedge \mathcal{I}_\varepsilon(\pi) = \mathsf{True} \implies \mathcal{AH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$$

PROOF:
ASSUME: 1. $\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi)$
2. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$
PROVE: $\mathcal{AH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$

The proof proceeds by case analysis on the symbolic rules for $\mathcal{AH}$.

1. CASE: **[Add Handler: Found]**
   1.1. $\exists \hat{e}' \in \mathtt{dom}(\hat{h})$ [Add Handler - Found (Symbolic)]
   1.2. $\hat{h}' = \hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') +\!\!+ [f]\right]$ [Add Handler - Found (Symbolic)]
   1.3. $\pi = (\hat{e} = \hat{e}')$ [Add Handler - Found (Symbolic)]
   1.4. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
   1.5. $\mathcal{I}_\varepsilon(\hat{e} = \hat{e}') = \mathsf{True}$ [Assumption 2 and 1.3]
   1.6. LET : $h = \mathcal{I}_\varepsilon(\hat{h})$

1.7. $e \in \texttt{dom}(h)$ [Assumption 1, 1.1 and 1.5]

1.8. $\mathcal{AH}(h, e, f) = h\,[e \mapsto h_o(e) +\!\!+ [f]]$ [Definition of $\mathcal{AH}$ (Concrete)]

1.9. $\mathcal{AH}(h, e, f) = h\,[e \mapsto h(e) +\!\!+ [f]]$ [1.7 and 1.8]

1.10. $h\,[e \mapsto h(e) +\!\!+ [f]] = \mathcal{I}_\varepsilon(\hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') +\!\!+ [f]\right])$ [Definition of $\mathcal{I}_\varepsilon$]

1.11. $\mathcal{AH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$ [1.2, 1.4 and 1.10]

□

2. CASE: **[Add Handler: Not Found]**

2.1. $\hat{e} \notin \texttt{dom}(\hat{h})$ [Add Handler - Not Found (Symbolic)]

2.2. $\hat{h}' = \hat{h}\,[\hat{e} \mapsto [f]]$ [Add Handler - Not Found (Symbolic)]

2.3. $\pi = \hat{e} \notin \texttt{dom}(\hat{h})$ [Add Handler - Not Found (Symbolic)]

2.4. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$

2.5. $\mathcal{I}_\varepsilon(\hat{e} \notin \texttt{dom}(\hat{h})) = \mathsf{True}$ [Assumption 2 and 2.3]

2.6. $e \notin \texttt{dom}(h)$ [Assumption 1, 2.1 and 2.5]

2.7. $\mathcal{AH}(h, e, f) = h\,[e \mapsto h_o(e) +\!\!+ [f]]$ [definition of $\mathcal{AH}$ (Concrete)]

2.8. $\mathcal{AH}(h, e, f) = h\,[e \mapsto [f]]$ [2.6 and 2.7]

2.9. $h\,[e \mapsto [f]] = \mathcal{I}_\varepsilon(\hat{h}\,[\hat{e} \mapsto [f]])$ [Definition of $\mathcal{I}_\varepsilon$]

2.10. $\mathcal{AH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$ [2.2, 2.4 and 2.9]

□

Lemma 6 (Remove Handler - Symbolic to Concrete)**.**

$$\mathcal{RH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi) \land \mathcal{I}_\varepsilon(\pi) = \mathsf{True} \implies \mathcal{RH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$$

PROOF:

ASSUME: 1. $\mathcal{RH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi)$

2. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$

PROVE: $\mathcal{RH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$

The proof proceeds by case analysis on the symbolic rules for $\mathcal{RH}$.

1. CASE: **[Remove Handler: Found]**

1.1. $\exists \hat{e}' \in \texttt{dom}(\hat{h})$ [Remove Handler - Found (Symbolic)]

1.2. $\hat{h}' = \hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') \setminus f\right]$ [Remove Handler - Found (Symbolic)]

1.3. $\pi = (\hat{e} = \hat{e}')$ [Remove Handler - Found (Symbolic)]

1.4. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$

1.5. $\mathcal{I}_\varepsilon(\hat{e} = \hat{e}') = \mathsf{True}$ [Assumption 2 and 1.3]

1.6. LET : $h = \mathcal{I}_\varepsilon(\hat{h})$

1.7. $e \in \texttt{dom}(h)$ [Assumption 1, 1.1 and 1.5]

1.8. $\mathcal{RH}(h, e, f) = h\,[e \mapsto h(e) \setminus f]$ [1.7 and definition of $\mathcal{RH}$]

1.9. $\mathcal{RH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}\left[\hat{e}' \mapsto \hat{h}(\hat{e}') \setminus f\right])$ [1.2 and 1.8]

1.10. $\mathcal{RH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$ [1.9 and definition of $\mathcal{RH}$]

□

2. CASE: **[Remove Handler: Not Found]**

2.1. $\hat{e} \notin \texttt{dom}(\hat{h})$ [Remove Handler - Not Found (Symbolic)]

2.2. $\hat{h}' = \hat{h}$ [Remove Handler - Not Found (Symbolic)]

2.3. $\pi = \hat{e} \notin \texttt{dom}(\hat{h})$ [Remove Handler - Not Found (Symbolic)]
2.4. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
2.5. $\mathcal{I}_\varepsilon(\hat{e} \notin \texttt{dom}(\hat{h})) = \mathsf{True}$ [Assumption 2 and 2.3]
2.6. $e \notin \texttt{dom}(h)$ [Assumption 1, 2.1 and 2.5]
2.7. $\mathcal{RH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$ [Definition of $\mathcal{RH}$, 2.2 and 2.6]
□

Lemma 7 (Find Handler - Symbolic to Concrete)**.**

$$\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow (\bar{\texttt{f}}, \pi) \wedge \mathcal{I}_\varepsilon(\pi) = \mathsf{True} \implies \mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = \bar{\texttt{f}}$$

PROOF:
ASSUME: 1. $\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow (\bar{\texttt{f}}, \pi)$
2. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$
PROVE: $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = \bar{\texttt{f}}$
The proof proceeds by case analysis on the symbolic rules for $\mathcal{FH}$.
1. CASE: **[Find Handler: Found]**
1.1. $\exists \hat{e}' \in \texttt{dom}(\hat{h})$ [Find Handler - Found (Symbolic)]
1.2. $\bar{\texttt{f}} = \hat{h}(\hat{e}')$ [Find Handler - Found (Symbolic)]
1.3. $\pi = (\hat{e} = \hat{e}')$ [Find Handler - Found (Symbolic)]
1.4. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
1.5. $\mathcal{I}_\varepsilon(\hat{e} = \hat{e}') = \mathsf{True}$ [Assumption 2 and 1.3]
1.6. LET : $h = \mathcal{I}_\varepsilon(\hat{h})$
1.7. $e \in \texttt{dom}(h)$ [Assumption 1, 1.1 and 1.5]
1.8. $\mathcal{FH}(h, e) = h_o(e)$ [1.7 and definition of $\mathcal{FH}$]
1.9. $\mathcal{FH}(h, e) = h(e)$ [1.2 and 1.8]
1.10. $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = \bar{\texttt{f}}$ [1.9 and definition of $\mathcal{FH}$]
□

2. CASE: **[Find Handler: Not Found]**
2.1. $\hat{e} \notin \texttt{dom}(\hat{h})$ [Find Handler - Not Found (Symbolic)]
2.2. $\bar{\texttt{f}} = []$ [Find Handler - Not Found (Symbolic)]
2.3. $\pi = \hat{e} \notin \texttt{dom}(\hat{h})$ [Find Handler - Not Found (Symbolic)]
2.4. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
2.5. $\mathcal{I}_\varepsilon(\hat{e} \notin \texttt{dom}(\hat{h})) = \mathsf{True}$ [Assumption 2 and 2.3]
2.6. $e \notin \texttt{dom}(h)$ [Assumption 1, 2.1 and 2.5]
2.7. $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = h_o(e) = []$ [Definition of $\mathcal{FH}$, 2.2 and 2.6]
2.8. $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = \bar{\texttt{f}}$ [2.7 and definition of $\mathcal{FH}$]
□

Lemma 8 (Mapping E-semantics and UL Concretely)**.**

$$\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c' \ \wedge\ \epsilon\alpha = \cdot\ \wedge\ \neg \mathrm{L}.\mathsf{final}(lc) \implies \exists \mathrm{p}, lc' \cdot lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$$

PROOF:The proof follows by case analysis on the E-semantics symbolic rules.

Assume: 1. $\langle lc, h, q\rangle \leadsto_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$

2. $\epsilon\alpha = \cdot$

3. $\neg \mathrm{L}.\mathsf{final}(lc)$

Prove: $\exists \mathrm{p}, lc' \cdot lc \leadsto_{\mathrm{L}}^{\mathrm{p}} lc'$

1. Case: **[Environment Event Dispatch]**
   1.1. $\epsilon\alpha = (e, v)$ [Environment Event Dispatch Rule (Concrete)]
   1.2. $\bot$ [Assumption 2 and 1.1]
   This rule is not applicable due to a contradiction. □

2. Case: **[Continuation - Success]**
   2.1. $\mathrm{L}.\mathsf{final}(lc)$ [Continuation - Sucess (Concrete)]
   2.2. $\bot$ [Assumption 3 and 2.1]
   This rule is not applicable due to a contradiction. □

3. Case: **[Continuation - Failure]**
   3.1. $\mathrm{L}.\mathsf{final}(lc)$ [Continuation - Sucess (Concrete)]
   3.2. $\bot$ [Assumption 3 and 3.1]
   This rule is not applicable due to a contradiction. □

For the remaining cases, the proof follows directly from the rule definition. It is always the case that $\exists \mathrm{p}, lc' \cdot lc \leadsto_{\mathrm{L}}^{\mathrm{p}} lc'$ holds. □

Definition A.1 (Correctness Criteria - UL-Symbolic Semantics).

$$\textsc{L-Directed-Soundness}$$
$$\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{lc}')) \wedge (\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc}) \wedge lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc' \implies (\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$$

$$\textsc{L-Directed-Completeness}$$
$$\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{lc}')) \wedge (\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc}) \implies \exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'$$

Theorem A.1 (Directed Soundness of the Symbolic E-semantics).

$$\widehat{\epsilon c} \rightsquigarrow^{\hat{\epsilon\alpha}}_{\hat{\mathsf{E}}} \widehat{\epsilon c}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')) \wedge (\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \wedge (\varepsilon, \epsilon\alpha) \in \mathcal{M}_\pi(\hat{\epsilon\alpha}) \wedge \epsilon c \rightsquigarrow^{\epsilon\alpha}_{\mathsf{E}} \epsilon c' \implies (\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}')$$

Proof:

Assume:
1. $\widehat{\epsilon c} \rightsquigarrow^{\hat{\epsilon\alpha}}_{\hat{\mathsf{E}}} \widehat{\epsilon c}'$
2. $\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')$
3. $(\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c})$
4. $(\varepsilon, \epsilon\alpha) \in \mathcal{M}_\pi(\hat{\epsilon\alpha})$
5. $\epsilon c \rightsquigarrow^{\epsilon\alpha}_{\mathsf{E}} \epsilon c'$

Prove: $(\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}')$

The proof follows by case analysis on the symbolic semantics rules.

1. Case: **[Language Transition]**
   - 1.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]
   - 1.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]
   - 1.3. $\hat{\epsilon\alpha} = \cdot$ [Language Transition Rule - Symbolic]
   - 1.4. $\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}'$ [Language Transition Rule - Symbolic]
   - 1.5. $\hat{\mathrm{p}} = \cdot$ [Language Transition Rule - Symbolic]
   - 1.6. $\widehat{\epsilon c}' = \langle \widehat{lc}', \hat{h}, \hat{q} \rangle$ [Language Transition Rule - Symbolic]
   - 1.7. $\epsilon c = \mathcal{I}_\varepsilon(\widehat{\epsilon c}) \wedge \mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [by Assumption 3 and definition of $\mathcal{M}()$]
   - 1.8. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\epsilon\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]
   - 1.9. $\epsilon\alpha = \cdot$ [by 1.8 and Definition of $\mathcal{I}_\varepsilon$]
   - 1.10. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 1.1]
   - 1.11. $\neg \mathrm{L}.\mathsf{final}(\widehat{lc})$ [by 1.4 and definition of $\mathsf{isFinal}$]
   - 1.12. $\neg \mathrm{L}.\mathsf{final}(lc)$ [by Lemma 3, given 1.10, 1.11 and definition of $\mathcal{M}_\pi()$]
   - 1.13. $\exists \mathrm{p}, lc'.\ lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'$ [by Lemma 8, given Assumption 5, 1.9 and 1.12]
   - 1.14. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Assumption 2 and Definition of $\mathsf{pc}()$]
   - 1.15. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 1.4, 1.10, 1.13 and 1.14]
   - 1.16. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 1.4, 1.10, 1.13 and 1.14]
   - 1.17. $\mathrm{p} = \cdot$ [by 1.3, 1.16, definition of $\mathcal{M}_\pi()$ and definition of $\mathcal{I}_\varepsilon$]
   - 1.18. $\epsilon c' = \langle \mathcal{I}_\varepsilon(\widehat{lc}'), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q}) \rangle$ [by 1.13, 1.17 and the Language Transition Rule - Concrete]
   - 1.19. $(\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 1.18 and definition of $\mathcal{M}()$]

   □

2. Case: **[Add Handler]**

2.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

2.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

2.3. $\epsilon\hat{\alpha} = \cdot$ [Add Handler Rule - Symbolic]

2.4. $\widehat{lc} \rightsquigarrow^{\hat{\text{p}}}_{\text{L}} \widehat{lc}'$ [Add Handler Rule - Symbolic]

2.5. $\hat{\text{p}} = \mathsf{addHdlr}\langle \hat{e}, f \rangle$ [Add Handler Rule - Symbolic]

2.6. $\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi_{ah})$ [Add Handler Rule - Symbolic]

2.7. Let : $\widehat{lc}'' = \text{L}.\mathsf{assume}(\widehat{lc}', \pi_{ah})$

2.8. $\widehat{\epsilon c}' = \langle \widehat{lc}'', \hat{h}', \hat{q} \rangle$ [Add Handler Rule - Symbolic]

2.9. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 2.1]

2.10. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\epsilon\hat{\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]

2.11. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [by 2.10 and Set Theory]

2.12. $\epsilon\alpha = \cdot$ [by 2.3, 2.10 and definition of $\mathcal{I}_\varepsilon$]

2.13. $\neg\text{L}.\mathsf{final}(\widehat{lc})$ [by 2.4 and definition of $\mathsf{isFinal}$]

2.14. $\neg\text{L}.\mathsf{final}(lc)$ [by Lemma 3, given 2.9, 2.13 and definition of $\mathcal{M}_\pi()$]

2.15. $\exists \text{p}, lc'.\ lc \rightsquigarrow^{\text{p}}_{\text{L}} lc'$ [by Lemma 8, given Assumption 5, 2.2, 2.12 and 2.2]

2.16. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}'')$ [by Assumption 2, 2.8 and definition of $\mathsf{pc}()$]

2.17. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Lemma 2, given 2.7 and 2.16]

2.18. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 2.4, 2.9, 2.15 and 2.17]

2.19. $lc' = \mathcal{I}_\varepsilon(\widehat{lc}')$ [by 2.18 and definition of $\mathcal{M}_\pi()$]

2.20. $(\varepsilon, \text{p}) \in \mathcal{M}_\pi(\hat{\text{p}})$ [by Definition A.1, given 2.4, 2.9, 2.15 and 2.17]

2.21. Let : $e = \mathcal{I}_\varepsilon(\hat{e})$

2.22. $\text{p} = \mathsf{addHdlr}\langle e, f \rangle$ [by 2.5, 2.20, 2.21 and definition of $\mathcal{I}_\varepsilon(\mathsf{addHdlr}\langle \hat{e}, f \rangle)$]

2.23. Let : $h' = \mathcal{AH}(h, e, f)$

2.24. $\epsilon c' = \langle lc', h', q \rangle$ [Add Handler Rule - Concrete]

2.25. $\pi \Rightarrow \pi_{ah}$ [by Lemma 2, given 2.7 and 2.16]

2.26. $\mathcal{I}_\varepsilon(\mathsf{pc}(\widehat{lc}'')) = \mathsf{True}$ [by 2.11, 2.16 and definition of $\mathcal{I}_\varepsilon$]

2.27. $lc' = \mathcal{I}_\varepsilon(\widehat{lc}'')$ [by Lemma 3, given 2.7 and 2.26]

2.28. $\mathcal{AH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$ [by Lemma 5, given 2.6, 2.11 and 2.25]

2.29. $\mathcal{I}_\varepsilon(\hat{h}') = h'$ [by 2.23 and 2.28]

2.30. $\mathcal{I}_\varepsilon(\hat{q}) = q$ [by Assumption 2 and definition of $\mathcal{M}_\pi()$]

2.31. $\mathcal{I}_\varepsilon(\langle \widehat{lc}'', \hat{h}', \hat{q} \rangle) = \langle lc', h', q \rangle$ [by 2.27, 2.29, 2.30 and definition of $\mathcal{I}_\varepsilon$]

2.32. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 2.24, 2.31 and definition of $\mathcal{M}_\pi()$]

□

3. Case: **[Remove Handler]**

3.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

3.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

3.3. $\epsilon\hat{\alpha} = \cdot$ [Remove Handler Rule - Symbolic]

3.4. $\widehat{lc} \rightsquigarrow^{\hat{\text{p}}}_{\text{L}} \widehat{lc}'$ [Remove Handler Rule - Symbolic]

3.5. $\hat{\text{p}} = \mathsf{remHdlr}\langle \hat{e}, f \rangle$ [Remove Handler Rule - Symbolic]

3.6. $\mathcal{RH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi_{rh})$ [Remove Handler Rule - Symbolic]

3.7. Let : $\widehat{lc}'' = \text{L}.\mathsf{assume}(\widehat{lc}', \pi_{rh})$

3.8. $\hat{\epsilon c}' = \langle \widehat{lc}'', \hat{h}', \hat{q} \rangle$ [Remove Handler Rule - Symbolic]

3.9. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 3.1]

3.10. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\epsilon\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]

3.11. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [by 3.10 and Set Theory]

3.12. $\epsilon\alpha = \cdot$ [by 3.3, 3.10 and definition of $\mathcal{I}_\varepsilon$]

3.13. $\neg \mathrm{L}.\mathsf{final}(\widehat{lc})$ [by 3.4 and definition of $\mathsf{isFinal}$]

3.14. $\neg \mathrm{L}.\mathsf{final}(lc)$ [by Lemma 3, given 3.9, 3.13 and definition of $\mathcal{M}_\pi()$]

3.15. $\exists \mathrm{p}, lc'.\ lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'$ [by Lemma 8, given Assumption 5, 3.2, 3.12 and 3.14]

3.16. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}'')$ [by Assumption 2, 3.8 and definition of $\mathsf{pc}()$]

3.17. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Lemma 2, given 3.7 and 3.16]

3.18. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 3.4, 3.9, 3.15 and 3.17]

3.19. $lc' = \mathcal{I}_\varepsilon(\widehat{lc}')$ [3.18 and definition of $\mathcal{M}_\pi()$]

3.20. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 3.4, 3.9, 3.15 and 3.17]

3.21. Let : $e = \mathcal{I}_\varepsilon(\hat{e})$

3.22. $\mathrm{p} = \mathsf{remHdlr}\langle e, f \rangle$ [by 3.5, 3.20, 3.21 and definition of $\mathcal{I}_\varepsilon(\mathsf{remHdlr}\langle \hat{e}, f \rangle)$]

3.23. Let : $h' = \mathcal{RH}(h, e, f)$

3.24. $\epsilon c' = \langle lc', h', q \rangle$ [Remove Handler Rule - Concrete]

3.25. $\pi \Rightarrow \pi_{rh}$ [by Lemma 2, given 3.7 and 3.16]

3.26. $\mathcal{I}_\varepsilon(\mathsf{pc}(\widehat{lc}'')) = \mathsf{True}$ [by 3.11, 3.17 and definition of $\mathcal{I}_\varepsilon$]

3.27. $lc' = \mathcal{I}_\varepsilon(\widehat{lc}'')$ [by Lemma 3, given 3.7, 3.11 and 3.26]

3.28. $\mathcal{RH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e}), f) = \mathcal{I}_\varepsilon(\hat{h}')$ [by Lemma 6, given 3.11, 3.25 and 3.27]

3.29. $\mathcal{I}_\varepsilon(\hat{h}') = h'$ [by 3.23 and 3.28]

3.30. $\mathcal{I}_\varepsilon(\hat{q}) = q$ [by Assumption 3 and definition of $\mathcal{M}_\pi()$]

3.31. $\mathcal{I}_\varepsilon(\langle \widehat{lc}'', \hat{h}', \hat{q} \rangle) = \langle lc', h', q \rangle$ [3.27, 3.29, 3.30 and definition of $\mathcal{I}_\varepsilon()$]

3.32. $\epsilon c' \in \mathcal{M}_\pi(\hat{\epsilon c}')$ [3.24, 3.31 and definition of $\mathcal{M}_\pi()$]

$\square$

4. Case: **[Synchronous Dispatch]**

4.1. $\hat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

4.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

4.3. $\hat{\epsilon\alpha} = \cdot$ [Synchronous Dispatch Rule - Symbolic]

4.4. $\widehat{lc} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}'$ [Synchronous Dispatch Rule - Symbolic]

4.5. $\hat{\mathrm{p}} = \mathsf{sDispatch}\langle \hat{e}, \hat{v} \rangle$ [Synchronous Dispatch Rule - Symbolic]

4.6. $\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi_{fh})$ [Synchronous Dispatch Rule - Symbolic]

4.7. Let : $\hat{q}' = [(f_i, \hat{e}, \hat{v}) \mid_{i=0}^n]$

4.8. Let : $\widehat{lc}'' = \mathrm{L}.\mathsf{assume}(\widehat{lc}', \pi)$

4.9. Let : $\widehat{lc}''' = \mathrm{L}.\mathsf{suspend}(\widehat{lc}'')$

4.10. $\hat{\epsilon c}' = \langle \widehat{lc}''', \hat{h}, \hat{q}' \mathbin{+\!\!+} [(\widehat{lc}'', (\lambda\widehat{lc}.\mathsf{True}))] \mathbin{+\!\!+} \hat{q} \rangle$ [Synchronous Dispatch Rule - Symbolic]

4.11. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 4.1]

4.12. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\epsilon\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]

4.13. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [by 4.12 and Set Theory]

4.14. $\epsilon\alpha = \cdot$ [by 4.3, 4.12 and definition of $\mathcal{I}_\varepsilon$]

4.15. $\neg \mathrm{L}.\mathsf{final}(\widehat{lc})$ [by 4.4 and definition of $\mathsf{isFinal}$]

4.16. $\neg$L.$\mathsf{final}(lc)$ [by Lemma 3, given 4.11, 4.15 and definition of $\mathcal{M}_\pi()$]
4.17. $\exists \text{p}, lc'.\ lc \rightsquigarrow^{\text{p}}_{\text{L}} lc'$ [by Lemma 8, given Assumption 5, 4.2, 4.14 and 4.16]
4.18. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}''')$ [Assumption 2, 4.10 and definition of $\mathsf{pc}()$]
4.19. $\pi \Rightarrow \mathsf{pc}(\text{L}.\mathsf{suspend}(\widehat{lc}''))$ [by 4.9 and 4.18]
4.20. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}'')$ [by Lemma 2, given 4.19]
4.21. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Lemma 2, given 4.8 and 4.20]
4.22. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 4.4, 4.11, 4.17 and 4.21]
4.23. $(\varepsilon, \text{p}) \in \mathcal{M}_\pi(\hat{\text{p}})$ [by Definition A.1, given 4.4, 4.11, 4.17 and 4.21]
4.24. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
4.25. LET : $v = \mathcal{I}_\varepsilon(\hat{v})$
4.26. $\text{p} = \mathsf{sDispatch}\langle e, v\rangle$ [4.5, 4.23, definition of $\mathcal{M}_\pi()$ and definition of $\mathcal{I}_\varepsilon(\mathsf{sDispatch}\langle \hat{e}, \hat{v}\rangle)$]
4.27. LET : $lc'' = \text{L}.\mathsf{suspend}(lc')$
4.28. LET : $[f_i \mid_0^n] = \mathcal{FH}(h, e)$
4.29. LET : $q' = [(f_i, [e, v]) \mid_{i=0}^n]$
4.30. $\epsilon c' = \langle lc'', h, q' +\!\!+ [(lc', (\lambda lc.\mathsf{True}))] +\!\!+ q\rangle$ [Synchronous Dispatch Rule - Concrete]
4.31. $\pi \Rightarrow \pi_{fh}$ [by Lemma 2, given 4.8 and 4.20]
4.32. $\mathcal{I}_\varepsilon(\mathsf{pc}(\widehat{lc}'')) = \mathsf{True}$ [by 4.13, 4.20 and definition of $\mathcal{I}_\varepsilon$]
4.33. $lc' = \mathcal{I}_\varepsilon(\widehat{lc}'')$ [by Lemma 3, given 4.8, 4.13 and 4.32]
4.34. $\mathcal{I}_\varepsilon(\text{L}.\mathsf{suspend}(\widehat{lc}'')) = \text{L}.\mathsf{suspend}(\mathcal{I}_\varepsilon(\widehat{lc}''))$ [by Lemma 3]
4.35. $\mathcal{I}_\varepsilon(\text{L}.\mathsf{suspend}(\widehat{lc}'')) = \text{L}.\mathsf{suspend}(lc')$ [by 4.33 and 4.34]
4.36. $(\varepsilon, lc'') \in \mathcal{M}_\pi(\widehat{lc}''')$ [by 4.9, 4.35 and definition of $\mathcal{M}_\pi()$]
4.37. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 2 and definition of $\mathcal{M}_\pi()$]
4.38. $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = [f_i \mid_0^n]$ [by Lemma 7, given 4.6, 4.13 and 4.31]
4.39. $\mathcal{I}_\varepsilon(\hat{q}' +\!\!+ [(\widehat{lc}'', (\lambda\widehat{lc}.\mathsf{True}))] +\!\!+ \hat{q}) = \mathcal{I}_\varepsilon(\hat{q}') +\!\!+ \mathcal{I}_\varepsilon([(\widehat{lc}'', (\lambda\widehat{lc}.\mathsf{True}))]) +\!\!+ \mathcal{I}_\varepsilon(\hat{q})$ [Definition of $\mathcal{I}_\varepsilon$]
4.40. $\mathcal{I}_\varepsilon(\hat{q}') = \mathcal{I}_\varepsilon([(f_i, \hat{e}, \hat{v}) \mid_{i=0}^n]) = [(\mathcal{I}_\varepsilon(f_i), e, v) \mid_{i=0}^n] = q'$ [by definition of $\mathcal{I}_\varepsilon$, 4.7, 4.29, 4.39]
4.41. $\mathcal{I}_\varepsilon([(\widehat{lc}'', (\lambda\widehat{lc}.\mathsf{True}))]) = [(lc', (\lambda lc.\mathsf{True}))]$ [by 4.33 and definition of $\mathcal{I}_\varepsilon$]
4.42. $\mathcal{I}_\varepsilon(\hat{q}) = q$ [by Assumption 2 and definition of $\mathcal{M}_\pi()$]
4.43. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 4.35, 4.39, 4.40, 4.41, 4.42 and definition of $\mathcal{M}_\pi()$]
□

5. CASE: **[Asynchronous Dispatch]**
5.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q}\rangle$ [Definition of symbolic E-configurations]
5.2. $\epsilon c = \langle lc, h, q\rangle$ [Definition of concrete E-configurations]
5.3. $\widehat{\epsilon\alpha} = \cdot$ [Asynchronous Dispatch Rule - Symbolic]
5.4. $\widehat{lc} \rightsquigarrow^{\hat{\text{p}}}_{\text{L}} \widehat{lc}'$ [Asynchronous Dispatch Rule - Symbolic]
5.5. $\hat{\text{p}} = \mathsf{aDispatch}\langle \hat{e}, \hat{v}\rangle$ [Asynchronous Dispatch Rule - Symbolic]
5.6. $\mathcal{FH}((\hat{h}, \hat{e})) \rightsquigarrow (([f_i \mid_0^n], \pi_{fh}))$ [Asynchronous Dispatch Rule - Symbolic]
5.7. LET : $\hat{q}' = [(f_i, \hat{e}, \hat{v}) \mid_{i=0}^n]$
5.8. LET : $\widehat{lc}'' = \text{L}.\mathsf{assume}(\widehat{lc}', \pi)$
5.9. $\widehat{\epsilon c}' = \langle \widehat{lc}'', \hat{h}, \hat{q}'\rangle$ [Asynchronous Dispatch Rule - Symbolic]
5.10. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 5.1]
5.11. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\widehat{\epsilon\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]
5.12. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [by 5.11 and Set Theory]

5.13. $\epsilon\alpha = \cdot$ [by 5.3, 5.11 and Definition of $\mathcal{I}_\varepsilon$]
5.14. $\neg$L.$\mathsf{final}(\widehat{lc})$ [by 5.4 and definition of $\mathsf{isFinal}$]
5.15. $\neg$L.$\mathsf{final}(lc)$ [by Lemma 3, given 5.10, 5.14 and definition of $\mathcal{M}_\pi()$]
5.16. $\exists \mathrm{p}, lc'.\ lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'$ [by Lemma 8, given Assumption 5, 5.2, 5.13 and 5.15]
5.17. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}'')$ [by Assumption 2, 5.9 and definition of $\mathsf{pc}()$]
5.18. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Lemma 2, given 5.8 and 5.17]
5.19. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 5.4, 5.10, 5.16 and 5.18]
5.20. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 5.4, 5.10, 5.16 and 5.18]
5.21. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
5.22. LET : $v = \mathcal{I}_\varepsilon(\hat{v})$
5.23. $\mathrm{p} = \mathsf{aDispatch}\langle e, v\rangle$ [by 5.5, 5.20, definition of $\mathcal{M}_\pi()$ and definition of $\mathcal{I}_\varepsilon(\mathsf{aDispatch}\langle \hat{e}, \hat{v}\rangle)$]
5.24. LET : $[f_i\ |_0^n] = \mathcal{FH}(h, e)$
5.25. LET : $q' = [(f_i, [e, v])\ |_{i=0}^n]$
5.26. $\epsilon c' = \langle lc', h, q'\rangle$ [Asynchronous Dispatch Rule - Concrete]
5.27. $\mathcal{I}_\varepsilon(\mathsf{pc}(\widehat{lc}'')) = \mathsf{True}$ [by 5.12, 5.17 and definition of $\mathcal{I}_\varepsilon$]
5.28. $lc' = \mathcal{I}_\varepsilon(\widehat{lc}'')$ [by Lemma 3, given 5.8 and 5.27]
5.29. $\pi \Rightarrow \pi_{fh}$ [by Lemma 2, given Assumption 2 and 5.8]
5.30. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}'')$ [by 5.12, 5.28 and definition of $\mathcal{M}_\pi()$]
5.31. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 3, 5.1, 5.2 and definition of $\mathcal{M}_\pi()$]
5.32. $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{e})) = [f_i\ |_0^n]$ [by Lemma 7, given 5.6, 5.12 and 5.29]
5.33. $\mathcal{I}_\varepsilon(\hat{q}') = \mathcal{I}_\varepsilon([(f_i, [\hat{e}, \hat{v}])\ |_{i=0}^n]) = [(f_i, [e, v])\ |_{i=0}^n] = q'$ [by 5.7 and definition of $\mathcal{I}_\varepsilon$]
5.34. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [5.9, 5.26, 5.30, 5.31, 5.33 and definition of $\mathcal{M}_\pi()$]
□

6. CASE: **[Environment Dispatch]**
6.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q}\rangle$ [Definition of symbolic E-configurations]
6.2. $\epsilon c = \langle lc, h, q\rangle$ [Definition of concrete E-configurations]
6.3. $\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i\ |_0^n], \pi_{fh})$ [Environment Dispatch Rule - Symbolic]
6.4. LET : $\hat{q}' = [(f_i, \hat{e}, \hat{v})\ |_{i=0}^n]$
6.5. LET : $\widehat{lc}' = \mathrm{L}.\mathsf{assume}(\widehat{lc}, \pi_{fh})$
6.6. $\widehat{\epsilon c}' = \langle \widehat{lc}', \hat{h}, \hat{q} +\!\!+ \hat{q}'\rangle$ [Environment Dispatch Rule - Symbolic]
6.7. $\epsilon\hat{\alpha} = (\hat{e}, \hat{v})$ [Environment Dispatch Rule - Symbolic]
6.8. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 6.1]
6.9. $\mathcal{I}_\varepsilon(\pi) = \mathsf{True}$ [by 6.8 and definition of $\mathcal{M}_\pi()$]
6.10. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
6.11. LET : $v = \mathcal{I}_\varepsilon(\hat{v})$
6.12. $\epsilon\alpha = (e, v)$ [by 6.7, 6.10, 6.11, Assumption 4 and definition of $\mathcal{M}_\pi(\epsilon\hat{\alpha})$]
6.13. $\epsilon c' = \langle lc, h, q +\!\!+ q'\rangle$ [Environment Dispatch Rule - Concrete]
6.14. $\mathcal{I}_\varepsilon(\hat{q}') = \mathcal{I}_\varepsilon([(f_i, [\hat{e}, \hat{v}])\ |_{i=0}^n]) = [(f_i, [e, v])\ |_{i=0}^n] = q'$ [by 6.4 and Definition of $\mathcal{I}_\varepsilon$]
6.15. $\mathcal{I}_\varepsilon(\hat{q} +\!\!+ \hat{q}') = \mathcal{I}_\varepsilon(\hat{q}) +\!\!+ \mathcal{I}_\varepsilon(\hat{q}')$ [by definition of $\mathcal{I}_\varepsilon$]
6.16. $\mathcal{I}_\varepsilon(\mathsf{pc}(\widehat{lc}')) = \mathsf{True}$ [by 6.5, 6.9 and definition of $\mathcal{I}_\varepsilon$]
6.17. $lc = \mathcal{I}_\varepsilon(\widehat{lc}')$ [by Lemma 3, given 6.5 and 6.16]
6.18. $\pi \Rightarrow \pi_{fh}$ [by Lemma 2, given Assumption 2 and 6.5]

6.19. $\mathcal{FH}(\mathcal{I}_\varepsilon(\hat{h}), e) = [f_i \mid_0^n]$ [by Lemma 7, given 6.3, 6.9 and 6.18]

6.20. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc}')$ [by 6.17 and definition of $\mathcal{M}_\pi()$]

6.21. $q' \in \mathcal{M}_\pi(\hat{q}')$ [by 6.14 and definition of $\mathcal{M}_\pi()$]

6.22. $\mathcal{I}_\varepsilon(\hat{q}) = q$ [by Assumption 2, 6.1, 6.2 and definition of $\mathcal{M}_\pi()$]

6.23. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 2, 6.1, 6.2 and definition of $\mathcal{M}_\pi()$]

6.24. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 6.6, 6.8, 6.13, 6.21, 6.22, 6.23 and definition of $\mathcal{M}_\pi()$]

□

7. Case: **[Schedule]**

7.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

7.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

7.3. $\widehat{\epsilon \alpha} = \cdot$ [Schedule Rule - Symbolic]

7.4. $\widehat{lc} \leadsto^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}'$ [Schedule Rule - Symbolic]

7.5. $\hat{\mathrm{p}} = \mathsf{schedule} f, \hat{v}$ [Schedule Rule - Symbolic]

7.6. Let : $\hat{q}' = \hat{q} +\!\!+ [(f, \hat{v})]$

7.7. $\widehat{\epsilon c}' = \langle \widehat{lc}', \hat{h}, \hat{q}' \rangle$ [Schedule Rule - Symbolic]

7.8. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 7.1]

7.9. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\widehat{\epsilon\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]

7.10. $\epsilon\alpha = \cdot$ [by 7.3, 7.9 and definition of $\mathcal{I}_\varepsilon$]

7.11. $\neg$L.$\mathsf{final}(\widehat{lc})$ [by 7.4 and definition of $\mathsf{isFinal}$]

7.12. $\neg$L.$\mathsf{final}(lc)$ [by Lemma 3, given 7.8, 7.11 and definition of $\mathcal{M}_\pi()$]

7.13. $\exists \mathrm{p}, lc'.\ lc \leadsto^{\mathrm{p}}_{\mathrm{L}} lc'$ [by Lemma 8, given Assumption 5, 7.2, 7.10 and 7.12]

7.14. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Assumption 2, 7.7, 7.8 and definition of $\mathsf{pc}()$]

7.15. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 7.4, 7.13 and 7.14]

7.16. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 7.4, 7.13 and 7.14]

7.17. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

7.18. $\mathrm{p} = \mathsf{schedule} f, v$ [by 7.5, 7.16, definition of $\mathcal{M}_\pi()$ and definition of $\mathcal{I}_\varepsilon(\mathsf{schedule} f, \hat{v})$]

7.19. Let : $q' = q +\!\!+ [(f, v)]$

7.20. $\epsilon c' = \langle lc', h, q' \rangle$ [by 7.13, 7.17, 7.18 and Schedule Rule - Concrete]

7.21. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 3, 7.1, 7.2 and definition of $\mathcal{M}_\pi()$]

7.22. $\mathcal{I}_\varepsilon(\hat{q}') = \mathcal{I}_\varepsilon(\hat{q} +\!\!+ [(f, \hat{v})]) = q +\!\!+ [(f, v)] = q'$ [by Assumption 3, 7.6, 7.17, 7.19 and definition of $\mathcal{I}_\varepsilon$]

7.23. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 7.7, 7.15, 7.20, 7.21, 7.22 and definition of $\mathcal{M}_\pi()$]

□

8. Case: **[Await]**

8.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

8.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

8.3. $\widehat{lc} \leadsto^{\hat{\mathrm{p}}}_{\mathrm{L}} \widehat{lc}'$ [Await Rule - Symbolic]

8.4. $\widehat{\epsilon \alpha} = \cdot$ [Await Rule - Symbolic]

8.5. $\hat{\mathrm{p}} = \mathsf{await}\langle \hat{v}, \hat{\rho} \rangle$ [Await Rule - Symbolic]

8.6. Let : $(\widehat{lc}_r, \widehat{lc}_a) = \mathrm{L}.\mathsf{splitReturn}(\widehat{lc}', \hat{v})$

8.7. $\widehat{\epsilon c}' = \langle \widehat{lc}_r, \hat{h}, \hat{q} +\!\!+ [(\widehat{lc}_a, \hat{\rho})] \rangle$ [Await Rule - Symbolic]

8.8. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 2 and 8.1]

8.9. $(\varepsilon, \epsilon\alpha) \in \{(\varepsilon, \mathcal{I}_\varepsilon(\epsilon\hat{\alpha})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$ [by Assumption 4 and definition of $\mathcal{M}_\pi()$]

8.10. $\epsilon\alpha = \cdot$ [by 8.4, 8.9 and definition of $\mathcal{I}_\varepsilon$]

8.11. $\neg\mathrm{L}.\mathsf{final}(\widehat{lc})$ [by 8.3 and definition of $\mathsf{isFinal}$]

8.12. $\neg\mathrm{L}.\mathsf{final}(lc)$ [by Lemma 3, given 8.8, 8.11 and definition of $\mathcal{M}_\pi()$]

8.13. $\exists \mathrm{p}, lc'.\ lc \rightsquigarrow^{\mathrm{p}}_{\mathrm{L}} lc'$ [by Lemma 8, given Assumption 5, 8.2, 8.10 and 8.12]

8.14. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}_r)$ [by Assumption 2, 8.7 and definition of $\mathcal{M}_\pi()$]

8.15. $\pi \Rightarrow \mathsf{pc}(\widehat{lc}')$ [by Lemma 2, given 8.6 and 8.14]

8.16. $(\varepsilon, lc') \in \mathcal{M}_\pi(\widehat{lc}')$ [by Definition A.1, given 8.3, 8.8 and 8.16]

8.17. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 8.3, 8.8 and 8.16]

8.18. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

8.19. Let : $\rho = \mathcal{I}_\varepsilon(\hat{\rho})$

8.20. $\mathrm{p} = \mathsf{await}\langle v, \rho\rangle$ [by 8.5, 8.17 8.18 and 8.19 and definition of $\mathcal{I}_\varepsilon$]

8.21. Let : $(lc_r, lc_a) = \mathrm{L}.\mathsf{splitReturn}(lc', v)$

8.22. $(lc_r, lc_a) = (\mathcal{I}_\varepsilon(\widehat{lc}_r), \mathcal{I}_\varepsilon(\widehat{lc}_a))$ [by Lemma 3, given 8.6 and 8.21]

8.23. $\epsilon c' = \langle lc_r, h, q +\!\!+ [(lc_a, \rho)]\rangle$ [by Await Rule - Concrete, given 8.2, 8.10, 8.13, 8.20 and 8.21]

8.24. $\mathcal{I}_\varepsilon(\widehat{lc}_r) = lc_r$ [by 8.22 and Equality of Tuples

8.25. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 3, 8.1, 8.2 and definition of $\mathcal{M}_\pi()$]

8.26. $\mathcal{I}_\varepsilon(\hat{q}) = q$ [by Assumption 3, 8.1, 8.2 and definition of $\mathcal{M}_\pi()$]

8.27. $\mathcal{I}_\varepsilon(\hat{q} +\!\!+ [(\widehat{lc}_a, \hat{\rho})]) = \mathcal{I}_\varepsilon(\hat{q}) +\!\!+ \mathcal{I}_\varepsilon([\widehat{lc}_a, \hat{\rho}]) = q +\!\!+ [lc_a, \rho]$ [by 8.22, 8.26 and definition of $\mathcal{I}_\varepsilon$]

8.28. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 8.7, 8.23, 8.24, 8.25, 8.27 and definition of $\mathcal{M}_\pi()$]

□

9. Case: **[Continuation - Success]**

9.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q}\rangle$ [Definition of symbolic E-configurations]

9.2. $\epsilon c = \langle lc, h, q\rangle$ [Definition of concrete E-configurations]

9.3. $\mathrm{L}.\mathsf{final}(\widehat{lc})$ [Continuation Success Rule (Symbolic)]

9.4. $\hat{q} = \hat{\kappa} : \hat{q}'$ [Continuation Success Rule (Symbolic)]

9.5. Let : $\widehat{lc}'' = \mathcal{CW}_{\mathrm{L}}(\widehat{lc}, \hat{\kappa})$

9.6. $\widehat{\epsilon c}' = \langle \widehat{lc}'', \hat{h}, \hat{q}'\rangle$ [Continuation Success Rule (Symbolic)]

9.7. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 9.1]

9.8. $\mathrm{L}.\mathsf{final}(lc)$ [by Lemma 3, given 9.3, 9.7 and definition of $\mathcal{M}_\pi()$]

9.9. Case: $\hat{\kappa} = (\widehat{lc}', \hat{\rho})$

9.9.1. $\widehat{lc}'' = \mathrm{L}.\mathsf{mergeConfs}(\widehat{lc}, \widehat{lc}')$ [by definition of $\mathcal{CW}_{\mathrm{L}}$]

9.9.2. Let : $lc' = \mathcal{I}_\varepsilon(\widehat{lc}')$

9.9.3. Let : $q' = \mathcal{I}_\varepsilon(\hat{q}')$

9.9.4. $q = \mathcal{I}_\varepsilon(\hat{q})$ [by Assumption 2 and definition of $\mathcal{M}_\pi()$]

9.9.5. $q = \mathcal{I}_\varepsilon(\hat{\kappa} : \hat{q}') = \mathcal{I}_\varepsilon(\hat{\kappa}) : \mathcal{I}_\varepsilon(\hat{q}')$ [by 9.4, 9.9.4 and definition of $\mathcal{I}_\varepsilon$]

9.9.6. Let : $\kappa = \mathcal{I}_\varepsilon(\hat{\kappa})$

9.9.7. Let : $\rho = \mathcal{I}_\varepsilon(\hat{\rho})$

9.9.8. $\kappa = (lc', \rho)$ [by 9.9.2, 9.9.6, 9.9.7 and definition of $\mathcal{I}_\varepsilon$]

9.9.9. Let : $lc'' = \mathcal{CW}_{\mathrm{L}}(lc, \kappa)$

9.9.10. $lc'' = \mathrm{L}.\mathsf{mergeConfs}(lc, lc')$ [by 9.9.7 and definition of $\mathcal{CW}_{\mathrm{L}}$]

9.9.11. $\epsilon c' = \langle lc'', h, q'\rangle$ [Continuation - Sucess (Concrete)]

9.9.12. $\mathcal{I}_\varepsilon(\widehat{lc}'') = lc''$ [by Lemma 3, given 9.9.1 9.9.2 and 9.9.10]

9.9.13. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 3 and definition of $\mathcal{M}_\pi()$]

9.9.14. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 9.6, 9.9.3, 9.9.11, 9.9.12, 9.9.13 and definition of $\mathcal{M}_\pi()$]

□

9.10. Case: $\hat{\kappa} = (f, \hat{v})$

9.10.1. $\hat{\rho}(\widehat{lc}) = \mathsf{True}$ [Continuation Success Rule (Symbolic)]

9.10.2. $\widehat{lc}'' = \text{L}.\mathsf{initialConf}(\widehat{lc}, (f, \hat{v}))$ [by 9.10.1 and Definition of $\mathcal{CW}_\text{L}$]

9.10.3. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

9.10.4. Let : $q' = \mathcal{I}_\varepsilon(\hat{q}')$

9.10.5. $q = \mathcal{I}_\varepsilon(\hat{q})$ [by Assumption 2 and definition of $\mathcal{M}_\pi()$]

9.10.6. $q = \mathcal{I}_\varepsilon(\hat{\kappa} : \hat{q}') = \mathcal{I}_\varepsilon(\hat{\kappa}) : \mathcal{I}_\varepsilon(\hat{q}')$ [by 9.4, 9.10.5 and definition of $\mathcal{I}_\varepsilon$]

9.10.7. Let : $\kappa = \mathcal{I}_\varepsilon(\hat{\kappa})$

9.10.8. Let : $\rho = \mathcal{I}_\varepsilon(\hat{\rho})$

9.10.9. $\kappa = (f, v)$ [by 9.10.3, 9.10.7 and definition of $\mathcal{I}_\varepsilon$]

9.10.10. Let : $lc' = \mathcal{CW}_\text{L}(lc, \kappa)$

9.10.11. $\rho(lc) = \mathsf{True}$ [9.10.1, 9.10.8 and definition of $\mathcal{I}_\varepsilon$]

9.10.12. $lc' = \text{L}.\mathsf{initialConf}(f, v)$ [9.10.10, 9.10.11 and definition of $\mathcal{CW}_\text{L}$]

9.10.13. $\epsilon c' = \langle lc', h, q' \rangle$ [Continuation - Sucess (Concrete)]

9.10.14. $\mathcal{I}_\varepsilon(\widehat{lc}'') = lc'$ [by Lemma 3, given 9.10.2 and 9.10.12]

9.10.15. $\mathcal{I}_\varepsilon(\hat{h}) = h$ [by Assumption 3 and definition of $\mathcal{M}_\pi()$]

9.10.16. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [9.6, 9.10.4, 9.10.14, 9.10.15 and definition of $\mathcal{M}_\pi()$]

□

10. Case: **[Continuation - Failure]**

10.1. Let : $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$

10.2. Let : $\epsilon c = \langle lc, h, q \rangle$

10.3. $\text{L}.\mathsf{final}(\widehat{lc})$ [Continuation Failure Rule (Symbolic)]

10.4. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 1 and 10.1]

10.5. $\hat{q} = \hat{\kappa} : \hat{q}'$ [Continuation Failure Rule (Symbolic)]

10.6. $(\widehat{lc}, \hat{\kappa}) \notin \mathtt{dom}(\mathcal{CW}_\text{L})$ [Continuation Failure Rule (Symbolic)]

10.7. $\widehat{\epsilon c}' = \langle \widehat{lc}, \hat{h}, \hat{q}' +\!\!+ [\hat{\kappa}] \rangle$ [Continuation - Failure (Symbolic)]

10.8. $\text{L}.\mathsf{final}(lc)$ [by Lemma 3, given 10.3, 10.4 and definition of $\mathcal{M}_\pi()$]

10.9. Let : $\kappa = \mathcal{I}_\varepsilon(\hat{\kappa})$

10.10. Let : $q' = \mathcal{I}_\varepsilon(\hat{q}')$

10.11. $(lc, \kappa) \notin \mathtt{dom}(\mathcal{CW}_\text{L})$ [by 10.6, 10.9, 10.10 and definition of $\mathcal{I}_\varepsilon$]

10.12. $\epsilon c' = \langle lc, h, q' +\!\!+ [\kappa] \rangle$ [Continuation Failure Rule (Concrete)]

10.13. $q' +\!\!+ [\kappa] \in \mathcal{M}_\pi(\hat{q}' +\!\!+ [\hat{\kappa}])$ [by 10.9, 10.10 and definition of $\mathcal{M}_\pi()$]

10.14. $\epsilon c' \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [by 10.6, 10.7, 10.13 and definition of $\mathcal{M}_\pi()$] □

Theorem A.2 (Directed Completeness of the Symbolic E-semantics)**.**

$$\widehat{\epsilon c} \rightsquigarrow^{\widehat{\epsilon\alpha}}_{\hat{\mathsf{E}}} \widehat{\epsilon c}' \;\wedge\; \pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}') \;\wedge\; (\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \implies \exists\, \epsilon\alpha, \epsilon c'.\; \epsilon c \rightsquigarrow^{\epsilon\alpha}_{\mathsf{E}} \epsilon c'$$

Proof:

ASSUME: 1. $\widehat{\epsilon c} \rightsquigarrow_{\hat{\mathsf{E}}}^{\epsilon\hat{\alpha}} \widehat{\epsilon c}'$
2. $\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')$
3. $(\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c})$
PROVE: $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$

The proof follows by case analysis on the symbolic semantics rules.

1. CASE: **[Language Transition]**
   1.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]
   1.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]
   1.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Language Transition Rule (Symbolic)]
   1.4. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 1.1]
   1.5. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [by Assumption 2 and definition of $\mathsf{pc}()$]
   1.6. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 1.3, 1.4, 1.5]
   1.7. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 1.3, 1.4, 1.5 and 1.6]
   1.8. $\epsilon c \rightsquigarrow_{\mathsf{E}} \langle lc', h, q \rangle$ [Language Transition Rule (Concrete)]
   1.9. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 1.8]
   □

2. CASE: **[Add Handler]**
   2.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]
   2.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]
   2.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Add Handler Rule (Symbolic)]
   2.4. $\hat{\mathrm{p}} = \mathsf{addHdlr}\langle \hat{e}, f \rangle$ [Add Handler Rule (Symbolic)]
   2.5. $\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi_{ah})$ [Add Handler Rule (Symbolic)]
   2.6. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 2.1]
   2.7. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [by Assumption 2 and definition of $\mathsf{pc}()$]
   2.8. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 2.3, 2.4 and 2.5]
   2.9. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 2.3, 2.4, 2.5 and 2.8]
   2.10. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$
   2.11. $\mathrm{p} = \mathsf{addHdlr}\langle e, f \rangle$ [by 2.9, 2.10, 2.11 and definition of $\mathcal{I}_\varepsilon(\mathsf{addHdlr}\langle \hat{e}, f \rangle)$]
   2.12. $\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}} \langle lc', \mathcal{AH}(h, e, f), q \rangle$ [Add Handler Rule (Concrete)]
   2.13. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 2.12]
   □

3. CASE: **[Remove Handler]**
   3.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]
   3.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]
   3.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Remove Handler Rule (Symbolic)]
   3.4. $\hat{\mathrm{p}} = \mathsf{remHdlr}\langle \hat{e}, f \rangle$ [Remove Handler Rule (Symbolic)]
   3.5. $\mathcal{RH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi_{ah})$ [Remove Handler Rule (Symbolic)]
   3.6. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 3.1]
   3.7. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [by Assumption 2 and definition of $\mathsf{pc}()$]
   3.8. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 3.3, 3.4 and 3.5]
   3.9. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 3.3, 3.4, 3.5 and 3.8]
   3.10. LET : $e = \mathcal{I}_\varepsilon(\hat{e})$

3.11. $\mathrm{p} = \mathsf{remHdlr}\langle e, f\rangle$ [by 3.9, 3.10, 3.11 and definition of $\mathcal{I}_\varepsilon(\mathsf{remHdlr}\langle \hat{e}, f\rangle)$]

3.12. $\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', \mathcal{RH}(h, e, f), q\rangle$ [Remove Handler Rule (Concrete)]

3.13. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 3.12]

□

4. Case: **[Synchronous Dispatch]**

4.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q}\rangle$ [Definition of symbolic E-configurations]

4.2. $\epsilon c = \langle lc, h, q\rangle$ [Definition of concrete E-configurations]

4.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Synchronous Dispatch Rule (Symbolic)]

4.4. $\hat{\mathrm{p}} = \mathsf{sDispatch}\langle \hat{e}, \hat{v}\rangle$ [Synchronous Dispatch Rule (Symbolic)]

4.5. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 4.1]

4.6. Let : $e = \mathcal{I}_\varepsilon(\hat{e})$

4.7. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

4.8. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [by Assumption 3 and definition of $\mathsf{pc}()$]

4.9. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 4.3, 4.5, 4.8]

4.10. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1 , given 4.3, 4.5, 4.8 and 4.9]

4.11. $\mathrm{p} = \mathsf{sDispatch}\langle e, v\rangle$ [by 4.4, 4.6, 4.7 and definition of $\mathcal{I}_\varepsilon(\mathsf{sDispatch}\langle \hat{e}, \hat{v}\rangle)$]

4.12. Let : $[f_i \mid_0^n] = \mathcal{FH}(h, e)$

4.13. Let : $q' = [(f_i, [e, v]) \mid_{i=0}^n]$

4.14. Let : $lc'' = \mathrm{L}.\mathsf{suspend}(lc')$

4.15. $\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc'', h, q' + [(lc', (\lambda lc.\mathsf{True}))] + q\rangle$ [Synchronous Dispatch Rule (Concrete)]

4.16. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 4.15]

□

5. Case: **[Asynchronous Dispatch]**

5.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q}\rangle$ [Definition of symbolic E-configurations]

5.2. $\epsilon c = \langle lc, h, q\rangle$ [Definition of concrete E-configurations]

5.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Asynchronous Dispatch Rule (Symbolic)]

5.4. $\hat{\mathrm{p}} = \mathsf{aDispatch}\langle \hat{e}, \hat{v}\rangle$ [Asynchronous Dispatch Rule (Symbolic)]

5.5. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 5.1]

5.6. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [Assumption 2 and definition of $\mathsf{pc}()$]

5.7. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 5.3, 5.5 and 5.6]

5.8. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 5.3, 5.5, 5.6 and 5.7]

5.9. Let : $e = \mathcal{I}_\varepsilon(\hat{e})$

5.10. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

5.11. $\mathrm{p} = \mathsf{aDispatch}\langle e, v\rangle$ [by 5.8, 5.9, 5.10 and definition of $\mathcal{I}_\varepsilon(\mathsf{aDispatch}\langle \hat{e}, \hat{v}\rangle)$]

5.12. Let : $[f_i \mid_0^n] = \mathcal{FH}(h, e)$

5.13. Let : $q' = [(f_i, [e, v]) \mid_{i=0}^n]$

5.14. $\langle lc, h, q\rangle \rightsquigarrow_{\mathsf{E}} \langle lc', h', q + q'\rangle$ [Asynchronous Dispatch Rule (Concrete)]

5.15. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 5.14]

□

6. Case: **[Environment Dispatch]**

6.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q}\rangle$ [Definition of symbolic E-configurations]

6.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

6.3. Let : $[f_i \mid_0^n] = \mathcal{FH}(h, e)$

6.4. Let : $q' = [(f_i, [e, v]) \mid_{i=0}^n]$

6.5. $\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}}^{(e,v)} \langle lc, h, q +\!\!+ q' \rangle$ [Environment Dispatch Rule (Concrete)]

6.6. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 6.5]

∎

7. Case: **[Schedule]**

7.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

7.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

7.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Schedule Rule (Symbolic)]

7.4. $\hat{\mathrm{p}} = \mathsf{schedule} f, \hat{v}$ [Schedule Rule (Symbolic)]

7.5. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 7.1]

7.6. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [Assumption 3 and definition of $\mathsf{pc}()$]

7.7. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 7.3, 7.5 and 7.6]

7.8. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 7.3, 7.5, 7.6 and 7.7]

7.9. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

7.10. $\mathrm{p} = \mathsf{schedule} f, v$ [by 7.4, 7.8, 7.9 and definition of $\mathcal{I}_\varepsilon(\mathsf{schedule} f, \hat{v})$]

7.11. Let : $q' = q +\!\!+ [(f, v)]$

7.12. $\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}} \langle lc', h, q' \rangle$ [Schedule Rule (Concrete)]

7.13. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 7.13]

∎

8. Case: **[Await]**

8.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

8.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

8.3. $\widehat{lc} \rightsquigarrow_{\mathrm{L}}^{\hat{\mathrm{p}}} \widehat{lc}'$ [Await Rule (Symbolic)]

8.4. $\hat{\mathrm{p}} = \mathsf{await}\langle \hat{v}, \hat{\rho} \rangle$ [Await Rule (Symbolic)]

8.5. $(\varepsilon, lc) \in \mathcal{M}_\pi(\widehat{lc})$ [by Lemma 4, given Assumption 3 and 8.1]

8.6. $\pi \Rightarrow \mathsf{pc}(\widehat{lc})$ [Assumption 3 and definition of $\mathsf{pc}()$]

8.7. $\exists\, \mathrm{p}, lc'.\ lc \rightsquigarrow_{\mathrm{L}}^{\mathrm{p}} lc'$ [by Definition A.1, given 8.3, 8.5, 8.6]

8.8. $(\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [by Definition A.1, given 8.3, 8.5, 8.6 and 8.8]

8.9. Let : $v = \mathcal{I}_\varepsilon(\hat{v})$

8.10. Let : $\rho = \mathcal{I}_\varepsilon(\hat{\rho})$

8.11. $\mathrm{p} = \mathsf{await}\langle v, \rho \rangle$ [by 8.4, 8.8, 8.9 and definition of $\mathcal{I}_\varepsilon(\mathsf{await}\langle \hat{v}, \hat{\rho} \rangle)$]

8.12. Let : $(lc_r, lc_a) = \mathrm{L}.\mathsf{splitReturn}(lc', v)$

8.13. $\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}} \langle lc_r, h, q +\!\!+ [(lc_a, \rho)] \rangle$ [Await Rule (Concrete)]

8.14. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 8.13]

∎

9. Case: **[Continuation - Success]**

9.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

9.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

9.3. $\epsilon\alpha = \cdot$ [Continuation - Success Rule (Symbolic)]

9.4. L.final($\widehat{lc}$) [Continuation - Success Rule (Symbolic)]

9.5. L.final(lc) [by Lemma 3, given Assumption 2, 9.4 and definition of $\mathcal{M}_\pi()$]

9.6. LET : $q = \kappa : q'$

9.7. $\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}} \langle \mathcal{CW}_{\mathrm{L}}(lc, \kappa), h, q' \rangle$ [Continuation - Success (Concrete)]

9.8. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [by 9.7]

□

10. CASE: **[Continuation - Failure]**

10.1. $\widehat{\epsilon c} = \langle \widehat{lc}, \hat{h}, \hat{q} \rangle$ [Definition of symbolic E-configurations]

10.2. $\epsilon c = \langle lc, h, q \rangle$ [Definition of concrete E-configurations]

10.3. $\epsilon\alpha = \cdot$ [Continuation - Failure Rule (Symbolic)]

10.4. $(\widehat{lc}, \hat{\kappa}) \notin \mathtt{dom}(\mathcal{CW}_{\mathrm{L}})$ [Continuation - Failure Rule (Symbolic)]

10.5. L.final($\widehat{lc}$) [Continuation - Failure Rule (Symbolic)]

10.6. L.final(lc) [by Lemma 3, given Assumption 2, 10.5 and definition of $\mathcal{M}_\pi()$]

10.7. LET : $q = \kappa : q'$

10.8. $(lc, \kappa) \notin \mathtt{dom}(\mathcal{CW}_{\mathrm{L}})$ [by Assumption 2, definition of $\mathcal{M}_\pi()$ and 10.4]

10.9. $\langle lc, h, q \rangle \rightsquigarrow_{\mathsf{E}} \langle \mathcal{CW}_{\mathrm{L}}(lc, \kappa), h, q' \rangle$ [Continuation - Success (Concrete)]

10.10. $\exists\, \epsilon\alpha, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{E}}^{\epsilon\alpha} \epsilon c'$ [10.8]

□

B. Message-Passing Semantics

B.1. Concrete Semantics

Values	Variables	E-confs	E-conf Ids	Ports	Messages
$v \in \mathcal{V}$	$x \in \mathcal{X}$	$\epsilon c \in \mathcal{EC}$	$\alpha \in \mathcal{A} \subset \text{Int}$	$p \in \mathcal{P} \subset \text{Int}$	$m \in \mathcal{M} := (vs, ps)$

Message-passing Primitives

$$\text{p} \in \text{P} := \cdot \mid \mathsf{send}\langle vs, ps, p_1, p_2\rangle \mid \mathsf{create}\langle x, vs\rangle \mid \mathsf{terminate}\langle\alpha\rangle \mid \mathsf{newPort}\langle\rangle \mid \mathsf{connect}\langle p_1, p_2\rangle \mid \mathsf{disconnect}\langle p\rangle \mid \mathsf{getConnected}\langle x, p\rangle \mid \mathsf{notifyAll}\langle v, vs\rangle \mid \mathsf{beginAtomic} \mid \mathsf{endAtomic} \mid \mathsf{fire}\langle v, vs\rangle$$

E-Conf Sequences	Message Queues	Port-confs Map	Conn-ports Map
$cs \in \mathcal{CS} : \overline{\mathcal{EC} \times \mathcal{A}}$	$mq \in \mathcal{MQ} : \overline{\mathcal{M} \times \mathcal{P}}$	$pcm \in \mathcal{PCM} : \mathcal{P} \rightharpoonup \mathcal{A}$	$cpm \in \mathcal{CPM} : \mathcal{P} \rightharpoonup \overline{\mathcal{P}}$

Lead Confs	MP-Configurations
$\ell \in \mathcal{L} := \cdot \mid \mathtt{Conf}\langle\alpha\rangle$	$mc \in \mathcal{MC} : \mathcal{CS} \times \mathcal{MQ} \times \mathcal{PCM} \times \mathcal{CPM} \times \mathcal{L}$

Configuration Actions

$$ca \in \mathcal{CA} := \cdot \mid \mathtt{Add}\langle\epsilon c, \alpha\rangle \mid \mathtt{Rem}\langle\alpha\rangle \mid \mathtt{Hold}\langle\alpha\rangle \mid \mathsf{Free}\langle\alpha\rangle \mid \mathsf{Notify}\langle v, vs\rangle$$

Figure B.1.: Message-Passing Syntax (Concrete)

Concrete Event Semantics Interface

1. $\mathsf{newConf}(vs)$: creates a new configuration based on the arguments vs that could be defined, for instance, as a tuple $\langle lc, h, q\rangle$, as defined by our E-semantics.
2. $\mathsf{setVar}(\epsilon c, x, v)$: updates the value of the variable x to v in the configuration ϵc.
3. $\mathsf{final}(\epsilon c)$: checks wether the event configuration ϵc is final. Intuitively, a configuration is final if there is nothing else to execute at the underlying language configuration.

Auxiliary Functions of the MP-semantics

Final: $\mathsf{final}(cs)$ returns true if all the event configurations in cs are final and false otherwise. To know whether a configuration is final, we make use of the underlying $\mathsf{final}(\epsilon c)$ function provided by the E-semantics;

Delete ports: $\mathtt{del\_ports}(ps, mq, pcm, cpm)$ deletes the ports of ps from mq, pcm and cpm;

Connect ports: $\mathtt{connect\_ports}(p_1, p_2, cpm)$ connects ports p_1 and p_2 in cpm;

Disconnect port: $\mathtt{disconnect\_port}(p, cpm)$ disconnects port p in cpm;

Transfer: $\mathtt{transfer}(\alpha, ps, pcm)$ transfers each port of ps to configuration α in pcm;

Final Configurations

$$\mathsf{final}(cs) \triangleq \begin{cases} \mathsf{true}, & \text{if } cs \text{ is empty} \\ \mathsf{ES.final}(c) \wedge \mathsf{final}(cs'), & \text{if } cs = c : cs' \end{cases}$$

Delete Ports

$$\texttt{del\_ports}(ps, mq, pcm, cpm) \triangleq (mq', pcm', cpm'), \text{ where } \begin{cases} mq' = mq \setminus ps \\ pcm' = pcm \setminus ps \\ cpm = cpm \setminus ps \end{cases}$$

Connect Ports

$$\texttt{connect\_ports}(p_1, p_2, cpm) \triangleq cpm', \text{ where} \begin{cases} ps_1 = cpm(p_1) \\ ps_2 = cpm(p_2) \\ cpm' = cpm[p_1 \mapsto ps_1 +\!\!+ [p_2], p_2 \mapsto ps_2 +\!\!+ [p_1]] \end{cases}$$

Disconnect Port

$$\texttt{disconnect\_port}(p, cpm) \triangleq cpm', \text{ where } cpm' = cpm \setminus p$$

Transfer Ports

$$\texttt{transfer}(\alpha, ps, pcm) \triangleq pcm', \text{ where} \begin{cases} ps = [p_i \mid_{i=0}^{n}] \\ pcm' = pcm[p_0 \mapsto \alpha, ..., p_n \mapsto \alpha] \end{cases}$$

Apply Config Action

$$\mathsf{applyAction}(cs, \ell, ca) \triangleq \begin{cases} (cs, \ell), & \text{if } ca \text{ is } \cdot \\ (cs +\!\!+ [\epsilon c_\alpha], \ell), & \text{if } ca \text{ is } \texttt{Add}\langle \epsilon c_\alpha \rangle \\ (cs \setminus \alpha, \ell), & \text{if } ca \text{ is } \texttt{Rem}\langle \alpha \rangle \\ (cs, \texttt{Conf}\langle \alpha \rangle), & \text{if } ca \text{ is } \texttt{Hold}\langle \alpha \rangle \\ (cs, \cdot), & \text{if} \begin{cases} ca \text{ is } \mathsf{Free}\langle \alpha \rangle \\ \ell \text{ is } \texttt{Conf}\langle \alpha \rangle \end{cases} \\ (cs', \ell), & \text{if } \begin{cases} ca \text{ is } \mathsf{Notify}\langle v, vs \rangle \\ cs = [\epsilon c_i \mid_{i=0}^{n}] \\ \epsilon c_i \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle v, vs \rangle} \epsilon c_i' \mid_{i=0}^{n} \\ cs' = [\epsilon c_i' \mid_{i=0}^{n}] \end{cases} \end{cases}$$

Apply Config Action: $\mathsf{applyAction}(cs, \ell, ca)$ updates the configuration queue cs and the lead configuration ℓ based on the configuration action ca. For instance, if the action is $\texttt{Add}\langle \epsilon c_\alpha \rangle$, the function adds the newly created configuration at the back of the configuration sequence cs and the lead configuration remans unchanged. In contrast, if the action is $\texttt{Hold}\langle \alpha \rangle$, the configuration sequence remains unchanged and the lead configuration becomes the one with identifier α.

Transition System (MP-semantics): $\langle cs, mq, pcm, cpm, \ell\rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm', \ell'\rangle$

[Run Configuration - Non Atomic Block] This rule is applied when there is no leading configuration, meaning that the semantics is not executing an atomic block. The scheduler chooses a configuration to run. The MP-semantics then applies the reduced-configuration transition to the chosen configuration and applies the resulting configuration action.

[Run Configuration - Atomic Block] In this case, the MP-semantics starts by obtaining the leading configuration ϵc_α. Then, it applies the reduced-configuration transition to ϵc_α proceeding analogously to the previous rule.

[Process Message] The scheduler can also choose to process a message from the message queue by returning $\mathsf{Msg}\langle((vs, ps), p), mq'\rangle$. The MP-semantics then processes the message by firing the ProcessMessage event with the arguments vs supplied in the message on the target configuration ϵc

$$\textsc{Run Conf - Non Atomic}\quad \frac{\begin{array}{c}\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, \epsilon c, cs_{post}\rangle \\ \langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca\rangle \\ cs', \ell' = \mathsf{applyAction}(cs_{pre} + [\epsilon c'] + cs_{post}, \cdot, ca)\end{array}}{\langle cs, mq, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm', \ell'\rangle}$$

$$\textsc{Run Conf - Atomic}\quad \frac{\begin{array}{c}cs_{pre} + [\epsilon c_\alpha] + cs_{post} = cs \quad \ell = \mathtt{Conf}\langle\alpha\rangle \\ \langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca\rangle \\ cs', \ell' = \mathsf{applyAction}(cs_{pre} + [\epsilon c'] + cs_{post}, \ell, ca)\end{array}}{\langle cs, mq, pcm, cpm, \ell\rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm', \ell'\rangle}$$

$$\textsc{Process Message}\quad \frac{\begin{array}{c}\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Msg}\langle((vs, ps), p), mq'\rangle \quad \alpha = pcm(p) \\ pcm' = \mathtt{transfer}(\alpha, ps, pcm) \quad cs_{pre} + [\epsilon c_\alpha] + cs_{post} = cs \\ \epsilon c_\alpha \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \textsc{ProcessMessage}, vs\rangle} \epsilon c'_\alpha \quad cs' = cs_{pre} + [\epsilon c'] + cs_{post}\end{array}}{\langle cs, mq, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq', pcm', cpm, \cdot\rangle}$$

Transition System (Scheduler) The scheduler either choses a configuration to run ($\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, \epsilon c, cs_{post}\rangle$) or a message to be processed ($\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Msg}\langle((vs, ps), p), mq'\rangle$). We provide an example below.

$$\textsc{Configuration Scheduled}\quad \frac{\begin{array}{c}cs_{pre} + [\epsilon c] + cs_{post} = cs \\ \mathsf{final}(cs_{pre}) \quad !\mathsf{ES.final}(\epsilon c)\end{array}}{\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, c, cs_{pos}\rangle}$$

$$\textsc{Message Scheduled}\quad \frac{\begin{array}{c}mq' = mq \triangleright (\lambda(vs, ps) \cdot ps \neq []) + mq \triangleright (\lambda(vs, ps) \cdot ps = []) \\ m :: mq'' = mq' \quad \mathsf{final}(cs)\end{array}}{\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Msg}\langle m, mq''\rangle}$$

Transition System (Reduced Semantics): $\langle \epsilon c, mq, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca\rangle$
In the following, we give the rules of the reduced semantics.

E-SEMANTICS TRANSITION

$$\frac{\epsilon c \rightsquigarrow^{\cdot}_{\mathsf{E}} \epsilon c'}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm\rangle}$$

The reduced semantics updates its inner configuration accordingly to the E-semantics transition.

POST MESSAGE

$$\frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{send}\langle vs, ps, p_1, p_2\rangle \qquad p_2 \in cpm(p_1) \quad mq' = mq +\!\!+ [((vs, ps), p_2)]}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm, cpm\rangle}$$

The reduced semantics enqueues the message (vs, ps) in the message queue

NEW EXECUTION

$$\frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{create}\langle x, vs\rangle \qquad \epsilon c''_{\alpha} = \mathsf{ES.newConf}(vs) \qquad \epsilon c''' = \mathsf{ES.setVar}(\epsilon c', x, \alpha) \quad ca = \mathtt{Add}\langle \epsilon c''_{\alpha}\rangle}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c''', mq, pcm, cpm, ca\rangle}$$

The reduced semantics makes use of the auxiliary function $\mathsf{newConf}(vs)$ for creating a fresh configuration.

TERMINATE EXECUTION

$$\frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{terminate}\langle \alpha\rangle \quad ps = pcm \triangleright \alpha \qquad (mq', pcm', cpm') = \mathtt{del\_ports}(ps, mq, pcm, cpm) \qquad ca = \mathtt{Rem}\langle \alpha\rangle}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq', pcm', cpm', ca\rangle}$$

The reduced semantics is responsible for immediately terminating the configuration with identifier α.

NEW PORT

$$\frac{\epsilon c_{\alpha} \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_{\alpha} \quad \mathrm{p} = \mathsf{newPort}\langle\rangle \quad p \text{ is fresh} \qquad pcm' = pcm[p \mapsto \alpha]}{\langle \epsilon c_{\alpha}, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_{\alpha}, mq, pcm', cpm\rangle}$$

The reduced semantics creates a fresh port in the current configuration.

GET CONNECTED PORTS

$$\frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{getConnected}\langle x, p\rangle \qquad ps = cpm(p) \quad \epsilon c'' = \mathsf{ES.setVar}(\epsilon c', x, ps)}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'', mq, pcm, cpm\rangle}$$

The reduced semantics obtains the ports ps connected with p by accessing the connected-ports map cpm.

CONNECT PORTS

$$\frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{connect}\langle p_1, p_2\rangle \qquad cpm' = \mathtt{connect\_ports}(p_1, p_2, cpm)}{\langle \epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm'\rangle}$$

The reduced semantics connects the ports p_1 and p_2 using the auxiliary function $\mathtt{connect\_ports}(p_1, p_2, cpm)$.

Disconnect Port

$$\frac{\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c' \quad \mathrm{p} = \mathsf{disconnect}\langle p \rangle \qquad cpm' = \mathtt{disconnect\_port}(p, cpm)}{\langle \epsilon c, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm' \rangle}$$

The reduced semantics disconnects p from all the ports to which it is currently connected with the help of the auxiliary function $\mathtt{disconnect\_port}(p, cpm)$.

Begin Atomic

$$\frac{\epsilon c_\alpha \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_\alpha \quad \mathrm{p} = \mathsf{beginAtomic} \qquad ca = \mathtt{Hold}\langle \alpha \rangle}{\langle \epsilon c_\alpha, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_\alpha, mq, pcm, cpm, ca \rangle}$$

The reduced semantics must ensure that there is no interleaving of configurations until an endAtomic primitive is found.

End Atomic

$$\frac{\epsilon c_\alpha \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_\alpha \quad \mathrm{p} = \mathsf{endAtomic} \qquad ca = \mathsf{Free}\langle \alpha \rangle}{\langle \epsilon c_\alpha, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_\alpha, mq, pcm, cpm, ca \rangle}$$

The reduced semantics generates the configuration action $\mathsf{Free}\langle \alpha \rangle$, indicating that the scheduler can run normally and the configuration α does not need to be chosen.

Notify All

$$\frac{\epsilon c_\alpha \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'_\alpha \quad \mathrm{p} = \mathsf{notifyAll}\langle v, vs \rangle \qquad ca = \mathsf{Notify}\langle v, vs \rangle}{\langle \epsilon c, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c', mq, pcm, cpm, ca \rangle}$$

The reduced semantics generates the configuration action $\mathsf{Notify}\langle v, vs \rangle$, so that the event v is triggered on all configurations with arguments vs.

B.2. Symbolic Semantics

Values	**Variables**	**E-confs**	**E-conf Ids**	**Ports**	**Messages**
$\hat{v} \in \hat{\mathcal{V}}$	$\hat{x} \in \hat{\mathcal{X}}$	$\widehat{\epsilon c} \in \widehat{\mathcal{EC}}$	$\alpha \in \mathcal{A} \subset \mathrm{Int}$	$p \in \mathcal{P} \subset \mathrm{Int}$	$\hat{m} \in \hat{\mathcal{M}} := (vs, ps)$

Message-passing Primitives

$\hat{\mathrm{p}} \in \hat{\mathrm{P}} := \cdot \mid \mathsf{send}\langle \hat{vs}, ps, p_1, p_2 \rangle \mid \mathsf{create}\langle \hat{x}, \hat{vs} \rangle \mid \mathsf{terminate}\langle \alpha \rangle \mid \mathsf{newPort}\langle\rangle \mid \mathsf{connect}\langle p_1, p_2 \rangle \mid \mathsf{disconnect}\langle p \rangle \mid \mathsf{getConnected}\langle \hat{x}, p \rangle \mid \mathsf{notifyAll}\langle \hat{v}, \hat{vs} \rangle \mid \mathsf{fire}\langle \hat{v}, \hat{vs} \rangle \mid \mathsf{beginAtomic} \mid \mathsf{endAtomic}$

E-Conf Sequences	**Message Queues**	**Port-confs Map**	**Conn-ports Map**
$\hat{cs} \in \hat{\mathcal{CS}} : \widehat{\mathcal{EC}} \times \mathcal{A}$	$\hat{mq} \in \hat{\mathcal{MQ}} : \hat{\mathcal{M}} \times \mathcal{P}$	$pcm \in \mathcal{PCM} : \mathcal{P} \rightharpoonup \mathcal{A}$	$cpm \in \mathcal{CPM} : \mathcal{P} \rightharpoonup \overline{\mathcal{P}}$

Lead Confs	**MP-Configurations**
$\ell \in \mathcal{L} := \cdot \mid \mathtt{Conf}\langle \alpha \rangle$	$\widehat{mc} \in \widehat{\mathcal{MC}} : \hat{\mathcal{CS}} \times \hat{\mathcal{MQ}} \times \mathcal{PCM} \times \mathcal{CPM} \times \mathcal{L}$

Configuration Actions

$\hat{ca} \in \widehat{\mathcal{CA}} := \cdot \mid \mathtt{Add}\langle \widehat{\epsilon c}, \alpha \rangle \mid \mathtt{Rem}\langle \alpha \rangle \mid \mathtt{Hold}\langle \alpha \rangle \mid \mathsf{Free}\langle \alpha \rangle \mid \mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle$

Figure B.2.: Message-Passing Syntax (differences to concrete execution highlighted in blue)

Symbolic E-semantics Interface

1. $\mathsf{newConf}(\hat{vs})$: creates a new configuration based on the arguments $\hat{vs}$ that could be defined, for instance, as a tuple $\langle \widehat{lc}, \hat{h}, \hat{q} \rangle$, as defined by our E-semantics.
2. $\mathsf{setVar}(\widehat{\epsilon c}, \hat{x}, \hat{v})$: updates the value of the variable $\hat{x}$ to $\hat{v}$ in the configuration $\widehat{\epsilon c}$.

Apply Config Action

$$\mathsf{applyAction}(\hat{cs}, \ell, \hat{ca}) \triangleq \begin{cases} (\hat{cs}, \ell), & \text{if } ca \text{ is } \cdot \\ (\hat{cs} + [\widehat{\epsilon c}_\alpha], \ell), & \text{if } ca \text{ is } \mathtt{Add}\langle \widehat{\epsilon c}_\alpha \rangle \\ (\hat{cs} \setminus \alpha, \ell), & \text{if } ca \text{ is } \mathtt{Rem}\langle \alpha \rangle \\ (\hat{cs}, \mathtt{Conf}\langle \alpha \rangle), & \text{if } ca \text{ is } \mathtt{Hold}\langle \alpha \rangle \\ (\hat{cs}, \cdot), & \text{if } \begin{cases} ca \text{ is } \mathsf{Free}\langle \alpha \rangle \\ \ell \text{ is } \mathtt{Conf}\langle \alpha \rangle \end{cases} \\ (\hat{cs}', \ell), & \text{if } \begin{cases} ca \text{ is } \mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle \\ \hat{cs} = [\widehat{\epsilon c}_i \mid_{i=0}^{n}] \\ \widehat{\epsilon c}_i \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \hat{v}, \hat{vs} \rangle} \widehat{\epsilon c}'_i \mid_{i=0}^{n} \\ \hat{cs}' = [\widehat{\epsilon c}'_i \mid_{i=0}^{n}] \end{cases} \\ ([\mathsf{ES.assume}(\widehat{\epsilon c}_i, \mathtt{f}) \mid_{i=0}^{n}], \ell), & \text{if } \begin{cases} ca \text{ is } \mathsf{Assume}\langle \mathtt{f} \rangle \\ \hat{cs} = [\widehat{\epsilon c}_i \mid_{i=0}^{n}] \end{cases} \end{cases}$$

3. $\mathsf{final}(\widehat{\epsilon c})$: checks wether the event configuration $\widehat{\epsilon c}$ is final. Intuitively, a configuration is final if there is nothing else to execute at the underlying language configuration.

4. $\mathsf{assume}(\widehat{\epsilon c}, \pi) = \widehat{\epsilon c}'$, where $\widehat{\epsilon c}'$ is obtained from $\widehat{\epsilon c}$ by extending its path condition with the formula π, if such an extension is satisfiable.

5. $\mathsf{pc}(\widehat{\epsilon c}) = \pi$, where π is the path condition computed in the current branch of configuration $\widehat{\epsilon c}$.

Auxiliary Functions of MP-semantics The only auxiliary function that has a different definition from its concrete counterpart is Apply Config Action. The others are identical to the concrete ones.

Transition System (Reduced Semantics): $\langle \widehat{\epsilon c}, \hat{mq}, pcm, cpm, \cdot \rangle \rightsquigarrow_{\mathsf{MP}} \langle \widehat{\epsilon c}', \hat{mq}', pcm', cpm', ca \rangle$ All rules for the symbolic reduced semantics analogous to their concrete counterparts. However, we have an additional rule, which is given below.

Assume

$$\frac{\widehat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\hat{\mathrm{p}}} \widehat{\epsilon c}' \quad \hat{\mathrm{p}} = \mathsf{assume}\langle \mathtt{f} \rangle \quad ca = \mathsf{Assume}\langle \mathtt{f} \rangle}{\langle \widehat{\epsilon c}, \hat{mq}, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \widehat{\epsilon c}', \hat{mq}, pcm, cpm, ca \rangle}$$

The reduced semantics generates the configuration action $\mathsf{Assume}\langle \mathtt{f} \rangle$, indicating that the formula $\mathtt{f}$ should be appended to the path conditions of each event configuration.

B.2.1. Correctness

Interpretation of MP-semantics Structures

$\mathcal{CS}$ - Empty
$\mathcal{I}_\varepsilon(\emptyset) \triangleq \emptyset$

$\mathcal{CS}$ - Composition
$\mathcal{I}_\varepsilon(\hat{cs}_1 \uplus \hat{cs}_2) \triangleq \mathcal{I}_\varepsilon(\hat{cs}_1) \uplus \mathcal{I}_\varepsilon(\hat{cs}_2)$

$\mathcal{CS}$ - Cell
$\mathcal{I}_\varepsilon([(\hat{\epsilon c}, \alpha)]) \triangleq [(\mathcal{I}_\varepsilon(\hat{\epsilon c}), \alpha)]$

$\mathcal{MQ}$ - Empty
$\mathcal{I}_\varepsilon([]) \triangleq []$

$\mathcal{MQ}$ - Non-Empty
$\mathcal{I}_\varepsilon(([\hat{v}_1, ..., \hat{v}_n], ps), p) \triangleq (([\mathcal{I}_\varepsilon(\hat{v}_1), ..., \mathcal{I}_\varepsilon(\hat{v}_n), ps), p)]$

MP Confs
$\mathcal{I}_\varepsilon(\langle \hat{cs}, \hat{mq}, pcm, cpm, \ell \rangle) \triangleq \langle \mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm, \ell \rangle$

MP primitive - Send
$\mathcal{I}_\varepsilon(\mathsf{send}\langle \hat{vs}, ps, p_1, p_2 \rangle) \triangleq \mathsf{send}\langle \mathcal{I}_\varepsilon(\hat{vs}), ps, p_1, p_2 \rangle$

MP primitive - Create
$\mathcal{I}_\varepsilon(\mathsf{create}\langle \hat{x}, \hat{vs} \rangle) \triangleq \mathsf{create}\langle \mathcal{I}_\varepsilon(\hat{x}), \mathcal{I}_\varepsilon(\hat{vs}) \rangle$

Configuration Actions (Add)
$\mathcal{I}_\varepsilon(\mathtt{Add}\langle \hat{\epsilon c}, \alpha \rangle) \triangleq \mathtt{Add}\langle \mathcal{I}_\varepsilon(\hat{\epsilon c}), \alpha \rangle$

Configuration Actions (Notify)
$\mathcal{I}_\varepsilon(\mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle) \triangleq \mathsf{Notify}\langle \mathcal{I}_\varepsilon(\hat{v}), \mathcal{I}_\varepsilon(\hat{vs}) \rangle$

Models of Symbolic MP-semantics Structures

E-Configurations
$\mathcal{M}_\pi(\hat{\epsilon c}) \triangleq \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\epsilon c})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$

Primitives
$\mathcal{M}_\pi(\hat{\mathrm{p}}) \triangleq \{(\varepsilon, \mathcal{I}_\varepsilon(\hat{\mathrm{p}})) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$

MP-Configurations
$\mathcal{M}_\pi(\langle \hat{cs}, \hat{mq}, pcm, cpm, \hat{\ell} \rangle) \triangleq \{(\varepsilon, \langle \mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm, \mathcal{I}_\varepsilon(\hat{\ell}) \rangle) \mid \mathcal{I}_\varepsilon(\pi) = \mathsf{True}\}$

Requirements 9 (Scheduler)**.** *The MP-semantics scheduler should satisfy the following properties:*

1. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Conf}\langle \hat{cs}_{pre}, \hat{\epsilon c}, \hat{cs}_{post} \rangle \implies$
   $\mathsf{schedule}(\mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})) \rightsquigarrow \mathsf{Conf}\langle \mathcal{I}_\varepsilon(\hat{cs}_{pre}), \mathcal{I}_\varepsilon(\hat{\epsilon c}), \mathcal{I}_\varepsilon(\hat{cs}_{post}) \rangle$

2. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Msg}\langle ((\hat{vs}, \hat{ps}), \hat{p}), \hat{mq}_{rem} \rangle \implies$
   $\mathsf{schedule}(\mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})) \rightsquigarrow \mathsf{Msg}\langle (\mathcal{I}_\varepsilon(\hat{vs}), \mathcal{I}_\varepsilon(\hat{ps})), \mathcal{I}_\varepsilon(\hat{p})), \mathcal{I}_\varepsilon(\hat{mq}_{rem}) \rangle$

Requirements 10 (E-semantics Interface Functions)**.** *The interface functions of the E-semantics must preserve path conditions, as follows:*

1. $\mathsf{newConf}(\hat{vs}) = \hat{\epsilon c} \implies \mathsf{newConf}(\mathcal{I}_\varepsilon(\hat{vs})) = \mathcal{I}_\varepsilon(\hat{\epsilon c})$

2. $\mathsf{setVar}(\hat{\epsilon c}, \hat{x}, \hat{v}) = \hat{\epsilon c}' \implies \mathsf{setVar}(\hat{\epsilon c}, \hat{x}, \hat{v}) = \mathcal{I}_\varepsilon(\hat{\epsilon c}')$

3. $\mathsf{final}(\hat{\epsilon c}) \implies \mathsf{final}(\mathcal{I}_\varepsilon(\hat{\epsilon c}))$

4. $\epsilon c = \mathcal{I}_\varepsilon(\hat{\epsilon c}) \wedge \hat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \hat{v}, \hat{vs} \rangle} \hat{\epsilon c}' \implies \epsilon c \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \mathcal{I}_\varepsilon(\hat{v}), \mathcal{I}_\varepsilon(\hat{vs}) \rangle} \mathcal{I}_\varepsilon(\hat{\epsilon c}')$

Lemma 11 (Final - Symbolic to Concrete)**.**

$$\mathsf{final}(\hat{cs}) \implies \mathsf{final}(\mathcal{I}_\varepsilon(\hat{cs}))$$

Proof:

Assume: 1. $\mathsf{final}(\hat{cs})$

Prove: $\mathsf{final}(\mathcal{I}_\varepsilon(\hat{cs}))$

The proof follows by induction on the length of the configuration sequence $\hat{cs}$.

1. Base Case: $\hat{cs}$ has length 0
   1.1. $\mathsf{final}(\mathcal{I}_\varepsilon(\hat{cs})) = \mathsf{final}(\mathcal{I}_\varepsilon([]))$ [Assumption 1 and step 1]
   1.2. $\mathsf{final}(\mathcal{I}_\varepsilon([])) = \mathtt{true}$ [Definition of final and step 1.1]
   ∎

2. Inductive Case: $\hat{cs}$ has length $n+1$. We assume that the property holds for length n and prove that it is valid for length $n+1$

   Assume: 1. $\mathsf{final}(\hat{cs}) \implies \mathsf{final}(\mathcal{I}_\varepsilon(\hat{cs}))$

   Prove: $\mathsf{final}([\widehat{\epsilon c}] +\!\!+ \hat{cs}) \implies \mathsf{final}(\mathcal{I}_\varepsilon([\widehat{\epsilon c}] +\!\!+ \mathcal{I}_\varepsilon(\hat{cs})))$

   2.1. $\mathsf{final}([\widehat{\epsilon c}] +\!\!+ \hat{cs}) = \mathsf{ES.final}(\widehat{\epsilon c}) \wedge \mathsf{final}(\hat{cs})$
   2.2. $\mathsf{ES.final}(\widehat{\epsilon c}) \wedge \mathsf{final}(\hat{cs}) \implies \mathsf{ES.final}(\mathcal{I}_\varepsilon(\widehat{\epsilon c})) \wedge \mathsf{final}(\hat{cs})$ [Assumption 10]
   2.3. $\mathsf{ES.final}(\mathcal{I}_\varepsilon(\widehat{\epsilon c})) \wedge \mathsf{final}(\hat{cs}) \implies \mathsf{ES.final}(\mathcal{I}_\varepsilon(\widehat{\epsilon c})) \wedge \mathsf{final}(\mathcal{I}_\varepsilon(\hat{cs}))$ [Step 2.2 and Assumption 1]
   2.4. $\mathsf{ES.final}(\mathcal{I}_\varepsilon(\widehat{\epsilon c})) \wedge \mathsf{final}(\hat{cs}) \implies \mathsf{final}(\mathcal{I}_\varepsilon([\widehat{\epsilon c}] +\!\!+ \hat{cs}))$ [Definition of final, $\mathcal{I}_\varepsilon$ and step 2.3]
   2.5. $\mathsf{final}([\widehat{\epsilon c}] +\!\!+ \hat{cs}) \implies \mathsf{final}(\mathcal{I}_\varepsilon([\widehat{\epsilon c}] +\!\!+ \hat{cs}))$ [Steps 2.1, 2.2, 2.3 and 2.4]
   ∎

Lemma 12 (Delete Ports - Symbolic to Concrete)**.**

$$\mathtt{del\_ports}(ps, \hat{mq}, pcm, cpm) \triangleq (\hat{mq}', pcm', cpm') \implies$$

$$\mathtt{del\_ports}(ps, \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm) \triangleq (\mathcal{I}_\varepsilon(\hat{mq}'), pcm', cpm')$$

Proof:

Assume: 1. $\mathtt{del\_ports}(ps, \hat{mq}, pcm, cpm) \triangleq (\hat{mq}', pcm', cpm')$

Prove: $\mathtt{del\_ports}(ps, \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm) \triangleq (\mathcal{I}_\varepsilon(\hat{mq}'), pcm', cpm')$

1. $\mathtt{del\_ports}(ps, \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm) = (\mathcal{I}_\varepsilon(\hat{mq}) \setminus ps, pcm \setminus ps, cpm \setminus ps)$ [Definition of del_ports]
2. $(\hat{mq}', pcm', cpm') = (\hat{mq} \setminus ps, pcm \setminus ps, cpm \setminus ps)$ [Definition of del_ports and Assumption 1]
3. $\hat{mq}' = \hat{mq} \setminus ps$ [Equality of tuples and step 2]
4. $\mathcal{I}_\varepsilon(\hat{mq}') = \mathcal{I}_\varepsilon(\hat{mq} \setminus ps)$ [Definition of $\mathcal{I}_\varepsilon$ and step 3]
5. $\mathtt{del\_ports}(ps, \mathcal{I}_\varepsilon(\hat{mq}), pcm, cpm) \triangleq (\mathcal{I}_\varepsilon(\hat{mq}'), pcm', cpm')$ [Steps 1, 2, 3 and 4]

∎

Lemma 13 (Apply Config Action - Symbolic to Concrete)**.**

$$\mathsf{applyAction}(\hat{cs}, \ell, \hat{ca}) \triangleq \hat{cs}', \ell' \implies$$

$$\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\hat{ca})) \triangleq \mathcal{I}_\varepsilon(\hat{cs}'), \ell'$$

Proof:The proof follows by case analysis on the type of configuration action $\hat{ca}$

Assume: 1. $\mathsf{applyAction}(\hat{cs}, \ell, \hat{ca}) \triangleq \hat{cs}', \ell'$

Prove: $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\hat{ca})) \triangleq \mathcal{I}_\varepsilon(\hat{cs}'), \ell'$

1. Case $\hat{ca} = \cdot$
   1.1. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\cdot)) = (\mathcal{I}_\varepsilon(\hat{cs}), \ell)$

1.2. $\ell = \ell'$ [Assumption 1 and Definition of $\mathsf{applyAction}$]

1.3. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\hat{ca})) \triangleq \mathcal{I}_\varepsilon(\hat{cs}'), \ell'$ [Steps 1.1 and 1.2]

□

2. CASE $\hat{ca} = \mathtt{Add}\langle \epsilon c_\alpha \rangle$

2.1. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\mathtt{Add}\langle \epsilon c_\alpha \rangle)) = (\mathcal{I}_\varepsilon(\hat{cs}) +\!\!+ ([\mathcal{I}_\varepsilon(\epsilon c_\alpha)], \ell)$

2.2. $(\hat{cs} +\!\!+ [\epsilon c_\alpha], \ell) = \hat{cs}', \ell'$ [Definition of $\mathsf{applyAction}$ and step 2.1]

2.3. $\hat{cs} +\!\!+ [\epsilon c_\alpha] = \hat{cs}' \wedge \ell = \ell'$ [Equality of tuples and step 2.2]

2.4. $\mathcal{I}_\varepsilon(\hat{cs} +\!\!+ [\epsilon c_\alpha]) = \mathcal{I}_\varepsilon(\hat{cs}')$ [Definition of $\mathcal{I}_\varepsilon$ and step 2.3]

2.5. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\hat{ca})) \triangleq \mathcal{I}_\varepsilon(\hat{cs}'), \ell'$ [Definition of $\mathsf{applyAction}$ and step 2.4]

□

3. CASE $\hat{ca} = \mathtt{Rem}\langle \alpha \rangle$

This case is analogous to $\mathtt{Add}\langle \epsilon c_\alpha \rangle$.

4. CASE $\hat{ca} = \mathtt{Hold}\langle \alpha \rangle$

4.1. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\mathtt{Hold}\langle \alpha \rangle)) = (\mathcal{I}_\varepsilon(\hat{cs}), \mathtt{Conf}\langle \alpha \rangle)$ [Definition of $\mathsf{applyAction}$]

4.2. $\mathsf{applyAction}(\hat{cs}, \ell, \mathcal{I}_\varepsilon(\mathtt{Hold}\langle \alpha \rangle)) = (\hat{cs}, \mathtt{Conf}\langle \alpha \rangle)$ [Definition of $\mathsf{applyAction}$]

4.3. $(\hat{cs}, \mathtt{Conf}\langle \alpha \rangle) = \hat{cs}', \ell'$ [Step 4.2 and Assumption 1]

4.4. $\hat{cs} = \hat{cs}' \wedge \mathtt{Conf}\langle \alpha \rangle = \ell'$ [Equality of tuples and step 4.3]

4.5. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\hat{ca})) \triangleq \mathcal{I}_\varepsilon(\hat{cs}'), \ell'$ [Steps 4.1, 4.2, 4.3 and 4.4]

□

5. CASE $\hat{ca} = \mathsf{Free}\langle \alpha \rangle$

This case is analogous to $\mathtt{Hold}\langle \alpha \rangle$.

6. CASE $\hat{ca} = \mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle$

6.1. LET : $\hat{cs} = [\widehat{\epsilon c}_i \mid_{i=0}^{n}]$

6.2. $\widehat{\epsilon c}_i \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \hat{v}, \hat{vs} \rangle} \widehat{\epsilon c}'_i \mid_{i=0}^{n}$ [Definition of $\mathsf{applyAction}$ and Assumption 1]

6.3. $\hat{cs}' = [\widehat{\epsilon c}'_i \mid_{i=0}^{n}]$ [Definition of $\mathsf{applyAction}$ and Assumption 1]

6.4. $\mathsf{applyAction}(\hat{cs}, \ell, \mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle) \triangleq \hat{cs}', \ell$ [Definition of $\mathsf{applyAction}$ and steps 6.2 and 6.3]

6.5. $\ell = \ell'$ [Assumption 1 and step 6.4]

6.6. $\mathcal{I}_\varepsilon(\hat{cs}) = [\mathcal{I}_\varepsilon(\epsilon c_i) \mid_{i=0}^{n}]$ [Definition of $\mathcal{I}_\varepsilon$ and step 6.1]

6.7. $\mathcal{I}_\varepsilon(\widehat{\epsilon c}_i) \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle \mathcal{I}_\varepsilon(\hat{v}), \mathcal{I}_\varepsilon(\hat{vs}) \rangle} \epsilon c'_i \mid_{i=0}^{n}$

6.8. LET : $cs' = [\epsilon c'_i \mid_{i=0}^{n}]$

6.9. $\mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}), \ell, \mathcal{I}_\varepsilon(\mathsf{Notify}\langle \hat{v}, \hat{vs} \rangle)) \triangleq cs', \ell$

6.10. $cs' = \mathcal{I}_\varepsilon(\hat{cs}')$ [Assumption 10 and steps 6.3, 6.1, 6.2, 6.3, 6.7]

□

□

Definition B.1 (Correctness Criteria - Symbolic E-semantics)**.**

E-DIRECTED-SOUNDNESS

$$\begin{array}{l} \widehat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\hat{\mathrm{p}}} \widehat{\epsilon c}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')) \wedge \\ (\varepsilon, \epsilon c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \wedge \epsilon c \rightsquigarrow_{\mathsf{E}}^{\mathrm{p}} \epsilon c' \\ \quad \Longrightarrow (\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}}) \end{array}$$

E-DIRECTED-COMPLETENESS

$$\begin{array}{l} \widehat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\hat{\mathrm{p}}} \widehat{\epsilon c}' \wedge (\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')) \wedge \\ (\varepsilon, c) \in \mathcal{M}_\pi(\widehat{\epsilon c}) \\ \quad \Longrightarrow \exists\, \mathrm{p}, c'.\ c \rightsquigarrow_{\mathsf{E}}^{\mathrm{p}} c' \end{array}$$

Theorem B.1 (Correctness of the Symbolic MP-semantics)**.**

$$\begin{array}{l}\textsc{MP-Directed-Soundness}\\ \widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}' \wedge \pi \Rightarrow \mathsf{pc}(\widehat{mc}') \wedge \\ \quad (\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc}) \wedge mc \rightsquigarrow_{\mathsf{MP}} mc' \\ \qquad \Longrightarrow (\varepsilon, mc') \in \mathcal{M}_\pi(\widehat{mc}')\end{array}$$

Proof:

Assume: 1. $\widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}'$

2. $\pi \Rightarrow \mathsf{pc}(\widehat{mc}')$

3. $(\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc})$

4. $mc \rightsquigarrow_{\mathsf{MP}} mc'$

Prove: $(\varepsilon, mc') \in \mathcal{M}_\pi(\widehat{mc}')$

The proof follows by case analysis on the MP-semantics rules.

1. Case: **[Run Conf - Non Atomic]**
   1.1. $\widehat{mc} = \langle \hat{cs}, \hat{mq}, pcm, cpm, \hat{\ell} \rangle$ [Definition of symbolic MP-configurations]
   1.2. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Conf}\langle \hat{cs}_{pre}, \hat{\epsilon c}, \hat{cs}_{post} \rangle$ [Run Conf - Non Atomic]
   1.3. $\langle \hat{\epsilon c}, \hat{mq}, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \hat{\epsilon c}', \hat{mq}', pcm', cpm', \hat{ca} \rangle$ [Run Conf - Non Atomic]
   1.4. $\hat{cs}', \hat{\ell}' = \mathsf{applyAction}(\hat{cs}_{pre} +\!\!+ [\hat{\epsilon c}'] +\!\!+ \hat{cs}_{post}, \cdot, \hat{ca})$ [Run Conf - Non Atomic]
   1.5. Case: **[E-semantics Transition]**
      1.5.1. $\hat{\epsilon c} \rightsquigarrow_{\mathsf{E}} \hat{\epsilon c}'$ [E-semantics Transition (symbolic)]
      1.5.2. $\langle \hat{\epsilon c}, \hat{mq}, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle \hat{\epsilon c}', \hat{mq}, pcm, cpm \rangle$ [E-semantics Transition]
      1.5.3. $\pi \Rightarrow \mathsf{pc}(\hat{\epsilon c}')$ [Definition of $\mathsf{pc}()$ and Assumption 2]
      1.5.4. $(\varepsilon, c) \in \mathcal{M}_\pi(\hat{\epsilon c})$ [Definition of $\mathcal{M}_\pi(\hat{\epsilon c})$ and Assumption 3]
      1.5.5. $c \rightsquigarrow_{\mathsf{E}} c'$ [E-semantics Transition (concrete) and Step 1.5.1]
      1.5.6. $(\varepsilon, c') \in \mathcal{M}_\pi(\hat{\epsilon c}')$ [Steps 1.5.1, 1.5.3, 1.5.4, 1.5.5 and Definition B.1]
      1.5.7. $\langle c', mq, pcm, cpm \rangle \in \mathcal{M}_\pi(\langle \hat{\epsilon c}', \hat{mq}, pcm, cpm \rangle)$ [Definition of $\mathcal{M}_\pi(rc)$, Step 1.5.6 and Assumption 3]
      1.5.8. $\hat{ca} = \cdot$
      1.5.9. $\hat{cs}' = \hat{cs}_{pre} +\!\!+ [\hat{\epsilon c}'] +\!\!+ \hat{c}spost$
      1.5.10. Let : $cs_{pre} = \mathcal{I}_\varepsilon(\hat{cs}_{pre}), cs_{post} = \mathcal{I}_\varepsilon(\hat{cs}_{post})$
      1.5.11. $\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, c, cs_{post} \rangle$ [Lemma 9 and Step 1.5.10]
      1.5.12. $\langle c, mq, pcm, cpm \rangle \rightsquigarrow_{\mathsf{MP}} \langle c', mq, pcm, cpm \rangle$ [E-semantics transition and Step 1.5.5]
      1.5.13. Let : $cs' = cs_{pre} +\!\!+ [c'] +\!\!+ cs_{post}$
      1.5.14. $\langle cs, mq, pcm, cpm, \cdot \rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq, pcm, cpm, \cdot \rangle$ [Assumption 4 and Step 1.5.13]
      1.5.15. $\langle cs', mq, pcm, cpm, \cdot \rangle \in \mathcal{M}_\pi(\langle \hat{cs}', \hat{mq}, pcm, cpm, \hat{\ell} \rangle)$ [Steps 1.5.6,1.5.9,1.5.13, 1.5.14 and defintion of $\mathcal{M}_\pi(\hat{\epsilon c})$]
   1.6. Case: **[New Execution]**
      1.6.1. $\hat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\hat{\mathrm{p}}} \hat{\epsilon c}'$ [E-semantics Transition (symbolic)]
      1.6.2. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Conf}\langle \hat{cs}_{pre}, \hat{\epsilon c}, \hat{cs}_{post} \rangle$ [Run Conf - Non Atomic]
      1.6.3. $\mathsf{schedule}(\mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})) \rightsquigarrow \mathsf{Conf}\langle \mathcal{I}_\varepsilon(\hat{cs}_{pre}), \mathcal{I}_\varepsilon(\hat{\epsilon c}), \mathcal{I}_\varepsilon(\hat{cs}_{post}) \rangle$ [Lemma 9 and step 1.6.2]
      1.6.4. $\hat{\mathrm{p}} = \mathsf{create}\langle \hat{x}, \hat{vs} \rangle$ [New Execution Rule (Symbolic)]
      1.6.5. Let : $\hat{\epsilon c}''_\alpha = \mathsf{ES.newConf}(\hat{vs})$
      1.6.6. Let : $\hat{\epsilon c}''' = \mathsf{ES.setVar}(\hat{\epsilon c}', \hat{x}, \alpha)$

1.6.7. LET : $\hat{ca} = \texttt{Add}\langle\widehat{\epsilon c}''_\alpha\rangle$

1.6.8. $\langle\widehat{\epsilon c}, \hat{mq}, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle\widehat{\epsilon c}''', \hat{mq}, pcm, cpm, \hat{ca}\rangle$ [NEW EXECUTION RULE (Symbolic) and steps 1.6.1, 1.6.4, 1.6.5, 1.6.6 and 1.6.7]

1.6.9. $\epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'$ [Concrete MP-semantics]

1.6.10. LET : $\epsilon c''_\alpha = \mathcal{I}_\varepsilon(\widehat{\epsilon c}''_\alpha)$

1.6.11. LET : $vs = \mathcal{I}_\varepsilon(\hat{vs})$

1.6.12. $\epsilon c''_\alpha = \mathsf{ES.newConf}(vs)$ [Assumption 10 and steps 1.6.5, 1.6.10 and 1.6.11]

1.6.13. $\epsilon c = \mathcal{I}_\varepsilon(\widehat{\epsilon c})$ [Assumption 3]

1.6.14. $(\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [Definition B.1, Assumption 2, steps 1.6.1, 1.6.9 and 1.6.10, and definition of $\mathcal{M}_\pi()$]

1.6.15. $\mathrm{p} = \texttt{Add}\langle\mathcal{I}_\varepsilon(\widehat{\epsilon c}''_\alpha)\rangle$ [Steps 1.6.7 and 1.6.14 and Definition of $\mathcal{M}_\pi()$]

1.6.16. $\mathrm{p} = \texttt{Add}\langle\epsilon c''_\alpha\rangle$ [Steps 1.6.10 and 1.6.15]

1.6.17. $\mathcal{I}_\varepsilon(\widehat{\epsilon c}''') = \mathsf{ES.setVar}(\mathcal{I}_\varepsilon(\widehat{\epsilon c}'), \mathcal{I}_\varepsilon(\hat{x}), \alpha)$ [Step 1.6.6 and Assumption 10]

1.6.18. LET : $\epsilon c''' = \mathcal{I}_\varepsilon(\widehat{\epsilon c}''')$, $x = \mathcal{I}_\varepsilon(\hat{x})$ and $ca = \mathcal{I}_\varepsilon(\hat{ca})$

1.6.19. $\langle\epsilon c, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle\epsilon c''', mq, pcm, cpm, ca\rangle$ [Concrete MP-semantics and steps 1.6.9, 1.6.12, 1.6.15, 1.6.16, 1.6.17 and 1.6.18]

1.6.20. LET : $cs_{pre} = \mathcal{I}_\varepsilon(\hat{cs}_{pre})$, $cs_{post} = \mathcal{I}_\varepsilon(\hat{cs}_{post})$

1.6.21. LET : $cs', \ell' = \mathsf{applyAction}(cs_{pre} +\!\!+ [\epsilon c'''] +\!\!+ cs_{post}, \ell, ca)$

1.6.22. $\langle cs, mq, pcm, cpm, \ell\rangle \rightsquigarrow_{\mathsf{MP}} \langle cs', mq, pcm, cpm, \ell'\rangle$ [Concrete MP-semantics]

1.6.23. LET : $\hat{cs}', \ell' = \mathsf{applyAction}(\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}'''] +\!\!+ \hat{cs}_{post}, \ell, \hat{ca})$

1.6.24. $\langle\hat{cs}, \hat{mq}, pcm, cpm, \ell\rangle \rightsquigarrow_{\mathsf{MP}} \langle\hat{cs}', \hat{mq}, pcm, cpm, \ell'\rangle$ [Symbolic MP-semantics and steps 1.6.8 and 1.6.23]

1.6.25. $\mathcal{I}_\varepsilon(\hat{cs}', \ell') = \mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}'''] +\!\!+ \hat{cs}_{post}), \ell, \mathcal{I}_\varepsilon(\hat{ca}))$ [Step 1.6.23 and Lemma 13]

1.6.26. $\mathcal{I}_\varepsilon(\hat{cs}'), \ell' = \mathsf{applyAction}(\mathcal{I}_\varepsilon(\hat{cs}_{pre}) +\!\!+ [\mathcal{I}_\varepsilon(\widehat{\epsilon c}''')] +\!\!+ \mathcal{I}_\varepsilon(\hat{cs}_{post}), \ell, \mathcal{I}_\varepsilon(\hat{ca}))$ [Definition of $\mathcal{I}_\varepsilon()$ and step 1.6.25]

1.6.27. $\mathcal{I}_\varepsilon(\hat{cs}'), \ell' = \mathsf{applyAction}(cs_{pre} +\!\!+ \epsilon c''' +\!\!+ cs_{post}, \ell, ca)$ [Step 1.6.2 and Assumption 3]

1.6.28. $\mathcal{I}_\varepsilon(\hat{cs}'), \ell' = cs', \ell'$ [Steps 1.6.20 and 1.6.27]

1.6.29. $\mathcal{I}_\varepsilon(\hat{cs}') = cs'$ [Step 1.6.28 and equality of tuples]

1.6.30. $\langle cs', mq, pcm, cpm, \ell'\rangle \in \mathcal{M}_\pi(\langle\hat{cs}', \hat{mq}, pcm, cpm, \hat{\ell}\rangle)$ [Definition of $\mathcal{M}_\pi()$, Assumption 3 and steps 1.6.24,1.6.29]

1.7. CASE: **[Begin Atomic]**

1.7.1. $\widehat{\epsilon c} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathsf{E}} \widehat{\epsilon c}'$ [SYMBOLIC MP-SEMANTICS (BEGIN ATOMIC)]

1.7.2. $\hat{\mathrm{p}} = \mathsf{beginAtomic}$ [SYMBOLIC MP-SEMANTICS (BEGIN ATOMIC)]

1.7.3. $\hat{ca} = \texttt{Hold}\langle\alpha\rangle$ [SYMBOLIC MP-SEMANTICS (BEGIN ATOMIC)]

1.7.4. $\langle\widehat{\epsilon c}_\alpha, \hat{mq}, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle\widehat{\epsilon c}'_\alpha, \hat{mq}, pcm, cpm, \hat{ca}\rangle$ [SYMBOLIC MP-SEMANTICS (BEGIN ATOMIC)]

1.7.5. $\hat{cs}', \hat{\ell}' = \mathsf{applyAction}(\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}'_\alpha] +\!\!+ \hat{cs}_{post}, \cdot, \hat{ca})$ [SYMBOLIC MP-SEMANTICS]

1.7.6. $\langle\hat{cs}, \hat{mq}, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle\hat{cs}', \hat{mq}, pcm, cpm, \ell'\rangle$ [SYMBOLIC MP-SEMANTICS]

1.7.7. LET : $cs_{pre}, cs_{post} = \mathcal{I}_\varepsilon(\hat{cs}_{pre}), \mathcal{I}_\varepsilon(\hat{cs}_{post})$

1.7.8. LET : $cs, mq = \mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})$

1.7.9. LET : $\epsilon c_\alpha = \mathcal{I}_\varepsilon(\widehat{\epsilon c}_\alpha)$

1.7.10. $\mathsf{schedule}(\mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, \epsilon c, \hat{cs}_{post}\rangle$ [Step 1.2 and Lemma 9]

1.7.11. $\exists \epsilon c' \cdot \epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'$ [CONCRETE MP-SEMANTICS, step 1.7.10 and Assumption 4]

1.7.12. $\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')$ [Definition of $\mathsf{pc}()$ and Assumption 2]

1.7.13. $(\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [Steps 1.7.1, 1.7.11, 1.7.12 and Definition B.1]

1.7.14. $\mathrm{p} = \mathsf{beginAtomic}$ [Steps 1.7.2 and 1.7.13]

1.7.15. LET : $ca = \mathtt{Hold}\langle\alpha\rangle$

1.7.16. $\langle \epsilon c_\alpha, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_\alpha, mq, pcm, cpm, ca\rangle$ [CONCRETE MP-SEMANTICS (BEGIN ATOMIC)]

1.7.17. LET : $cs', \ell' = \mathsf{applyAction}(cs_{pre} +\!\!+ [\epsilon c'_\alpha] +\!\!+ cs_{post}, \cdot, ca)$

1.7.18. $cs' = \mathcal{I}_\varepsilon(\hat{cs}')$ [Step 1.7.17 and Lemma 13]

1.7.19. $\langle cs', mq, pcm, cpm, \ell'\rangle \in \mathcal{M}_\pi(\langle \hat{cs}', \hat{mq}, pcm, cpm, \hat{\ell}\rangle)$ [Definition of $\mathcal{M}_\pi()$, Assumption 3 and step 1.7.18]

1.8. CASE: **[Post Message]**

1.8.1. $\widehat{\epsilon c} \rightsquigarrow^{\hat{\mathrm{p}}}_{\mathsf{E}} \widehat{\epsilon c}'$ [SYMBOLIC MP-SEMANTICS (POST MESSAGE)]

1.8.2. $\hat{\mathrm{p}} = \mathsf{send}\langle \hat{vs}, ps, p_1, p_2\rangle$ [SYMBOLIC MP-SEMANTICS (POST MESSAGE)]

1.8.3. $p_2 \in cpm(p_1)$ [SYMBOLIC MP-SEMANTICS (POST MESSAGE)]

1.8.4. LET : $\hat{mq}' = \hat{mq} +\!\!+ [((\hat{vs}, ps), p_2)]$ [SYMBOLIC MP-SEMANTICS (POST MESSAGE)]

1.8.5. $\langle \widehat{\epsilon c}, \hat{mq}, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \widehat{\epsilon c}', \hat{mq}', pcm, cpm\rangle$ [SYMBOLIC MP-SEMANTICS (POST MESSAGE)]

1.8.6. $\hat{ca} = \cdot$ [SYMBOLIC MP-SEMANTICS (POST MESSAGE)]

1.8.7. $\hat{cs}', \hat{\ell}' = \mathsf{applyAction}(\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}'_\alpha] +\!\!+ \hat{cs}_{post}, \cdot, \hat{ca})$ [SYMBOLIC MP-SEMANTICS (BEGIN ATOMIC)]

1.8.8. $\langle \hat{cs}, \hat{mq}, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle \hat{cs}', \hat{mq}', pcm, cpm, \ell'\rangle$ [SYMBOLIC MP-SEMANTICS]

1.8.9. LET : $cs_{pre}, cs_{post} = \mathcal{I}_\varepsilon(\hat{cs}_{pre}), \mathcal{I}_\varepsilon(\hat{cs}_{post})$

1.8.10. LET : $cs, mq = \mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})$

1.8.11. LET : $\epsilon c_\alpha = \mathcal{I}_\varepsilon(\widehat{\epsilon c}_\alpha)$

1.8.12. $\mathsf{schedule}(\mathcal{I}_\varepsilon(\hat{cs}), \mathcal{I}_\varepsilon(\hat{mq})) \rightsquigarrow \mathsf{Conf}\langle cs_{pre}, \epsilon c, \hat{cs}_{post}\rangle$ [Step 1.2 and Lemma 9]

1.8.13. $\exists \epsilon c' \cdot \epsilon c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} \epsilon c'$ [CONCRETE MP-SEMANTICS, step 1.8.12 and Assumption 4]

1.8.14. $\pi \Rightarrow \mathsf{pc}(\widehat{\epsilon c}')$ [Definition of $\mathsf{pc}()$ and Assumption 2]

1.8.15. $(\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}') \wedge (\varepsilon, \mathrm{p}) \in \mathcal{M}_\pi(\hat{\mathrm{p}})$ [Steps 1.8.1, 1.8.13, 1.8.14 and Definition B.1]

1.8.16. LET : $vs = \mathcal{I}_\varepsilon(\hat{vs})$

1.8.17. $\mathrm{p} = \mathsf{send}\langle vs, ps, p_1, p_2\rangle$ [Steps 1.8.2 and 1.8.16]

1.8.18. LET : $ca = \cdot$

1.8.19. LET : $mq' = mq +\!\!+ [((\hat{vs}, ps), p_2)]$

1.8.20. $\langle \epsilon c_\alpha, mq, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \epsilon c'_\alpha, mq', pcm, cpm, ca\rangle$ [Steps 1.8.16, 1.8.17, 1.8.18 and 1.8.19 and CONCRETE MP-SEMANTICS (POST MESSAGE)]

1.8.21. LET : $cs', \ell' = \mathsf{applyAction}(cs_{pre} +\!\!+ [\epsilon c'_\alpha] +\!\!+ cs_{post}, \cdot, ca)$

1.8.22. $cs' = \mathcal{I}_\varepsilon(\hat{cs}')$ [Step 1.8.21 and Lemma 13]

1.8.23. $mq' = \mathcal{I}_\varepsilon(\hat{mq}')$ [Definition of $\mathcal{M}_\pi()$ and steps 1.8.4 and 1.8.19]

1.8.24. $\langle cs', mq, pcm, cpm, \ell'\rangle \in \mathcal{M}_\pi(\langle \hat{cs}', \hat{mq}', pcm, cpm, \hat{\ell}\rangle)$ [Definition of $\mathcal{M}_\pi()$, Assumption 3 and steps 1.8.22 and 1.8.23]

1.9. CASE: **[Remaining Cases]**

The proof follows analogously to the previous cases.

2. CASE: **[Run Conf - Atomic]**

The proof follows analogously to Run Conf - Non Atomic

3. Case: **[Process Message]**
   - 3.1. $\widehat{\epsilon c} = \langle \hat{cs}, \hat{mq}, pcm, cpm, \cdot\rangle$ [Definition of symbolic MP-configurations and Process Message]
   - 3.2. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Msg}\langle((\hat{vs}, ps), p), \hat{mq}'\rangle$ [Process Message]
   - 3.3. $\alpha = pcm(p)$ [Process Message]
   - 3.4. $pcm' = \mathtt{transfer}(\alpha, ps, pcm)$ [Process Message]
   - 3.5. $\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}_\alpha] +\!\!+ \hat{cs}_{post} = \hat{cs}$ [Process Message]
   - 3.6. $v = $ ProcessMessage [Process Message]
   - 3.7. $\widehat{\epsilon c} \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle v, \hat{vs}\rangle} \widehat{\epsilon c}'$ [Process Message]
   - 3.8. $\langle \hat{cs}, \hat{mq}, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle \hat{cs}', \hat{mq}', pcm', cpm, \cdot\rangle$ [Assumption 4 and [Process Message]]
   - 3.9. Let : $vs = \mathcal{I}_\varepsilon(\hat{vs})$
   - 3.10. $c \rightsquigarrow_{\mathsf{E}}^{\mathsf{fire}\langle v, vs\rangle} c'$ [Assumption 4 and Step 3.9]
   - 3.11. Let : $c = \mathcal{I}_\varepsilon(\widehat{\epsilon c}), cs_{pre} = \mathcal{I}_\varepsilon(\hat{cs}_{pre}), cs_{post} = \mathcal{I}_\varepsilon(\hat{cs}_{post})$
   - 3.12. $cs \in \mathcal{M}_\pi(\hat{cs})$ [Definition of $\mathcal{M}_\pi(\widehat{\epsilon c})$ and Assumption 3]
   - 3.13. $cs = cs_{pre} +\!\!+ [c] +\!\!+ cs_{post}$ [Steps 3.9 and 3.12]
   - 3.14. Let : $cs' = cs_{pre} +\!\!+ [c'] +\!\!+ cs_{post}$
   - 3.15. $\mathsf{schedule}(cs, mq) \rightsquigarrow \mathsf{Msg}\langle((vs, ps), p), mq'\rangle$ [Lemma 9 and Step 3.2]
   - 3.16. $\epsilon c' = \langle cs, mq', pcm', cpm\rangle$ [Process Message and Steps 3.9, 3.10, 3.11, 3.12, 3.13, 3.14 and 3.15]
   - 3.17. $(\varepsilon, \epsilon c') \in \mathcal{M}_\pi(\widehat{\epsilon c}')$ [Definition of $\mathcal{M}_\pi(\widehat{\epsilon c})$ and Steps 3.12,3.14,3.15, 3.16]

   □

Theorem B.2 (Correctness of the Symbolic MP-semantics).

$$\begin{array}{l}\text{MP-Directed-Completeness}\\ \widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}' \wedge \pi \Rightarrow \mathsf{pc}(\widehat{mc}') \wedge \\ \quad (\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc}) \\ \quad\quad \implies \exists\, mc'.\ mc \rightsquigarrow_{\mathsf{MP}} mc'\end{array}$$

Proof:

Assume: 1. $\widehat{mc} \rightsquigarrow_{\mathsf{MP}} \widehat{mc}'$
2. $\pi \Rightarrow \mathsf{pc}(\widehat{mc}')$
3. $(\varepsilon, mc) \in \mathcal{M}_\pi(\widehat{mc})$

Prove: $\exists\, mc'.\ mc \rightsquigarrow_{\mathsf{MP}} mc'$

The proof follows by case analysis on the MP-semantics rules.

1. Case: **[Run Conf - Non Atomic]**
   - 1.1. $\widehat{mc} = \langle \hat{cs}, \hat{mq}, pcm, cpm, \hat{\ell}\rangle$
   - 1.2. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Conf}\langle \hat{cs}_{pre}, \widehat{\epsilon c}, \hat{cs}_{post}\rangle$ [Run Conf - Non Atomic]
   - 1.3. $\langle \widehat{\epsilon c}, \hat{mq}, pcm, cpm\rangle \rightsquigarrow_{\mathsf{MP}} \langle \widehat{\epsilon c}', \hat{mq}', pcm', cpm', \hat{ca}\rangle$ [Run Conf - Non Atomic]
   - 1.4. $\hat{cs}', \hat{\ell}' = \mathsf{applyAction}(\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}'] +\!\!+ \hat{cs}_{post}, \cdot, \hat{ca})$ [Run Conf - Non Atomic]
   - 1.5. Case: **[E-semantics Transition]**
     - 1.5.1. $\widehat{\epsilon c} \rightsquigarrow_{\mathsf{E}} \widehat{\epsilon c}'$ [E-semantics Transition (symbolic)]
     - 1.5.2. $\langle \widehat{\epsilon c}, \hat{mq}, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle \widehat{\epsilon c}', \hat{mq}, pcm, cpm, \cdot\rangle$ [E-semantics Transition (symbolic)]
     - 1.5.3. Let : $cs = \mathcal{I}_\varepsilon(\hat{cs}), mq = \mathcal{I}_\varepsilon(\hat{mq})$
     - 1.5.4. $\epsilon c = \langle cs, mq, pcm, cpm\rangle$ [Assumption 3, Step 1.5.3 and Definition of $\mathcal{M}_\pi(\widehat{\epsilon c})$]

1.5.5. $c \in \mathcal{M}_\pi(\widehat{\epsilon c})$ [Definition of $\mathcal{M}_\pi(\hat{cs})$ and Step 1.5.3]

1.5.6. $\exists\, \mathrm{p}, c'.\ c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} c'$ [Steps 1.5.1, Assumption 2, Step 1.5.5 and Definition B.1]

1.5.7. $c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} c'$ [Step 1.5.6]

1.5.8. $\langle c, mq, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle c', mq, pcm, cpm, \cdot\rangle$ [Step 1.5.7 and E-semantics Transition]

1.5.9. $\exists\, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{MP}} \epsilon c'$ [Step 1.5.8]

1.6. Case: **[Remaining Cases]**

The proof follows analogously to E-semantics Transition

2. Case: **[Run Conf - Atomic]**

The proof follows analogously to Run Conf - Non Atomic

3. Case: **[Process Message]**

3.1. $\widehat{\epsilon c} = \langle \hat{cs}, \hat{mq}, pcm, cpm, \cdot\rangle$ [Definition of symbolic MP-configurations and Process Message]

3.2. $\mathsf{schedule}(\hat{cs}, \hat{mq}) \rightsquigarrow \mathsf{Msg}\langle((\hat{vs}, ps), p), \hat{mq}'\rangle$ [Process Message]

3.3. $\alpha = pcm(p)$ [Process Message]

3.4. $pcm' = \mathtt{transfer}(\alpha, ps, pcm)$ [Process Message]

3.5. $\hat{cs}_{pre} +\!\!+ [\widehat{\epsilon c}_\alpha] +\!\!+ \hat{cs}_{post} = \hat{cs}$ [Process Message]

3.6. $v = \text{ProcessMessage}$ [Process Message]

3.7. $\widehat{\epsilon c} \rightsquigarrow^{\mathsf{fire}\langle v, \hat{vs}\rangle}_{\mathsf{E}} \widehat{\epsilon c}'$ [Process Message]

3.8. $\langle \hat{cs}, \hat{mq}, pcm, cpm, \cdot\rangle \rightsquigarrow_{\mathsf{MP}} \langle \hat{cs}', \hat{mq}', pcm', cpm, \cdot\rangle$ [Assumption 4 and [Process Message]]

3.9. Let : $cs = \mathcal{I}_\varepsilon(\hat{cs}), mq = \mathcal{I}_\varepsilon(\hat{mq})$

3.10. $\epsilon c = \langle cs, mq, pcm, cpm\rangle$ [Assumption 3, Step 3.9 and Definition of $\mathcal{M}_\pi(\widehat{\epsilon c})$]

3.11. $c \in \mathcal{M}_\pi(\widehat{\epsilon c})$ [Definition of $\mathcal{M}_\pi(\hat{cs})$, Step 3.9 and Assumption 3]

3.12. $\exists\, \mathrm{p}, c'.\ c \rightsquigarrow^{\mathrm{p}}_{\mathsf{E}} c'$ [Step 3.7, Assumption 2, Step 3.11 and Definition B.1]

3.13. $\exists\, \epsilon c'.\ \epsilon c \rightsquigarrow_{\mathsf{MP}} \epsilon c'$ [MP-semantics rules and Step 3.12]

□