A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications

Gabriela Sampaio
Imperial College London, United Kingdom
g.sampaio17@imperial.ac.uk

José Fragoso Santos
INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
Imperial College London, United Kingdom
jose.fragoso@tecnico.ulisboa.pt

Petar Maksimović
Imperial College London, United Kingdom
p.maksimovic@imperial.ac.uk

Philippa Gardner
Imperial College London, United Kingdom
p.gardner@imperial.ac.uk

Abstract

We introduce a trusted infrastructure for the symbolic analysis of modern event-driven Web applications. This infrastructure consists of reference implementations of the DOM Core Level 1, DOM UI Events, JavaScript Promises and the JavaScript `async`/`await` APIs, all underpinned by a simple Core Event Semantics which is sufficiently expressive to describe the event models underlying these APIs. Our reference implementations are trustworthy in that three follow the appropriate standards line-by-line and all are thoroughly tested against the official test-suites, passing all the applicable tests. Using the Core Event Semantics and the reference implementations, we develop JaVerT.Click, a symbolic execution tool for JavaScript that, for the first time, supports reasoning about JavaScript programs that use multiple event-related APIs. We demonstrate the viability of JaVerT.Click by proving both the presence and absence of bugs in real-world JavaScript code.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Software testing and debugging

Keywords and phrases Events, DOM, JavaScript, promises, symbolic execution, bug-finding

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.28

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.5`.

Funding Fragoso Santos, Gardner, and Maksimović were partially supported by the EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems" (EP/K008528/1) and the EPSRC Fellowship "VetSpec: Verified Trustworthy Software Specification" (EP/R034567/1).
Gabriela Sampaio: Sampaio was supported by a CAPES Foundation Scholarship, process number 88881.129599/2016-01.
José Fragoso Santos: Fragoso Santos was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), with reference UIDB/50021/2020 (INESC-ID multi-annual funding).

© Gabriela Sampaio, José Fragoso Santos, Petar Maksimović, and Philippa Gardner;
licensed under Creative Commons License CC-BY
34th European Conference on Object-Oriented Programming (ECOOP 2020).
Editors: Robert Hirschfeld and Tobias Pape; Article No. 28; pp. 28:1–28:29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

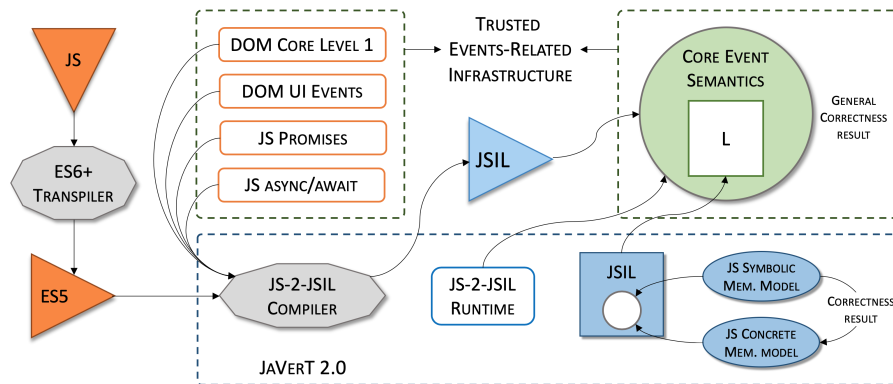


Figure 1 Infrastructure of JaVerT.Click.

1 Introduction

Event-driven programming lies at the core of modern Web applications, facilitated by a variety of APIs, such as DOM UI Events [53], JavaScript (JS) Promises [7] and Web Workers [52], each of which comes with its own event model and idiosyncrasies. There has been work on formalising and reasoning about some of these event models: e.g., Rajani et al. [29] have given a formal semantics of DOM UI Events, instrumented to disallow insecure information flows; Lerner at al. [19] have given a formal model and have proven several meta-properties of the DOM Event Dispatch algorithm; and Madsen et al. [22] have developed a calculus for reasoning about JS promises. In each case, the work targets a specific API and its associated event model, and it is not apparent how the work can be extended to include other APIs.

We introduce a trusted infrastructure for the symbolic analysis of modern event-driven Web applications which, we believe for the first time, supports reasoning about code that uses multiple event-related APIs within a single, unified formalism. This infrastructure comprises:

1. a Core Event Semantics, which identifies the fundamental building blocks underpinning the event models of widely-used APIs, and which is formalised and implemented parametrically, assuming an underlying language L (§2); and
2. trusted JS reference implementations of DOM Core Level 1, DOM UI Events, JS promises, and the JS `async/await` (§3-4), the APIs that we target in this paper.

Our infrastructure can readily be added on top of existing symbolic analysis tools; in this paper, we connect it to JaVerT 2.0 [13], a state-of-the-art symbolic analysis tool for JS, creating JaVerT.Click, the first symbolic analysis tool that can reason about JS programs that use multiple event-related APIs. We use JaVerT.Click to analyse `cash` [55] and `p−map` [35], two real-world JS libraries that interact with the targeted APIs, finding bugs in both and establishing bounded correctness of several important properties for `cash`.

The infrastructure of JaVerT.Click is illustrated in Figure 1. JaVerT.Click is built on top of JaVerT 2.0 [13], which supports three types of analysis: whole-program symbolic testing, verification, and automatic compositional testing based on bi-abduction; in this paper, we focus only on symbolic testing. The symbolic execution engine of JaVerT 2.0 works on JSIL, a simple intermediate language that can be instantiated with either the concrete or symbolic memory model of JS. JSIL comes with a correctness result that states that its symbolic testing has no false positives. JaVerT 2.0 targets the strict mode of the ECMAScript 5 standard (ES5 Strict), and comes with: JS-2-JSIL, a trusted compiler from ES5 Strict to JSIL which preserves the memory model and the semantics line-by-line, and is tested using the official Test262 test suite [6]; and the JS-2-JSIL runtime, which provides JSIL implementations of the ES5 Strict internal and built-in functions.

Our reference implementations are all written in ES5 Strict and get compiled to JSIL using JS-2-JSIL as part of JaVerT.Click. These implementations are trusted in that all except that of JS `async/await` (cf. §4.2) follow the API standards line-by-line and all are thoroughly tested against the official test suites, passing all the applicable tests. During the testing, we have discovered coverage gaps in the test suites of DOM Core Level 1 and UI Events and created additional tests to fill these gaps. Our choice to use JS as the API implementation language enables us to directly build on our previous JS analysis, simplifies implementations of promises and `async/await`, which rely on JS for some of their functionality, and makes the implementations easily reusable by other symbolic analysis tools for JS.

As our programs of interest use JS features beyond ES5 Strict, such as `async/await` and anonymous lambda-functions, we introduce a transpilation step from ES6+ Strict to ES5 Strict. This transpiled program and the compiled API reference implementations are then compiled to JSIL using the JS-2-JSIL compiler. The resulting JSIL code, together with the JS-2-JSIL runtime, is passed to the Core Event Semantics instantiated with either the JSIL concrete semantics (for testing) or the JSIL symbolic semantics (for analysis). Assuming correctness of the underlying language (e.g. JSIL), we give a general correctness result for the Core Event Semantics, proving that it has no false positives.

We apply JaVerT.Click to real-world JS code that calls the APIs studied in this paper (§5). In particular, we provide comprehensive symbolic testing of the `events` module of the `cash` library [55], a widely-used alternative for jQuery, which makes heavy use of DOM UI Events. We create a symbolic test suite for the `events` module with 100% line coverage, establishing bounded correctness of several important properties of the module, such as: ‘a handler can be executed for a given event if and only if it has been previously registered for that event’, and also discovering two subtle, previously unknown bugs. We also symbolically test the small, yet widely-used, `p−map` library [35], which uses JS promises and `async/await` to provide an extra layer of functionality on top of JS promises. We achieve 100% line coverage, discovering one bug. All discovered bugs have been reported and have since been fixed.

We believe that our infrastructure can straightforwardly be extended to support other event-driven Web APIs, such as File [51], postMessage [54], and Web Workers [52]. This would require a trusted JS reference implementation of the target API and, possibly, an extension of the Core Event Semantics with primitives that handle new types of event behaviour.

2 Core Event Semantics

Our ultimate goal is to develop a formalism within which one could reason symbolically about all event-related APIs. In this paper, we take an important step towards this goal by distilling the essence of three fundamental, complex such APIs – DOM UI events, JS promises, and JS `async/await` – into a minimal Core Event Semantics (onward: Event Semantics) that is easily extensible with support for further APIs. We define the Event 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. The Event Semantics interacts with the L-semantics by exposing a set of *labels*, which correspond to the fundamental operations underpinning the targeted APIs, such as event handler registration/deregistration and synchronous/asynchronous event dispatch. In this section, we first define the main concepts of the Event Semantics and explain the intuition behind them (§2.1), and then present the concrete (§2.2) and symbolic (§2.3) Event Semantics, connected with an appropriate correctness result.

Values	**Event Types**	**Function Ids**	**Handler Registers**	**L-Confs**	**Conf. Preds**
$v \in \mathcal{V}$	$e \in \mathcal{E} \subset \mathcal{V}$	$f \in \mathcal{F} \subset \mathcal{V}$	$h \in \mathcal{H} : \mathcal{E} \rightharpoonup \overline{\mathcal{F}}$	$c \in \mathcal{C}$	$p \in \mathcal{P} : \mathcal{C} \to \mathbb{B}$

Event Labels

$$\ell \in \mathcal{L} := \mathsf{addHdlr}\langle e, f\rangle \mid \mathsf{remHdlr}\langle e, f\rangle \mid \mathsf{sDispatch}\langle e, v\rangle \mid \mathsf{aDispatch}\langle e, v\rangle \mid \mathsf{schedule}\langle f, v\rangle \mid \mathsf{await}\langle v, p\rangle$$

Continuations	**Continuation Queues**	**E-Configurations**
$\kappa \in \mathcal{K} := (f, v) \mid (c, p)$	$q \in \mathcal{Q} : \overline{\mathcal{K}}$	$\omega \in \Omega : \mathcal{C} \times \mathcal{H} \times \mathcal{Q}$

Figure 2 Main Concepts of the Event Semantics.

2.1 Main Concepts of the Event Semantics

The main concepts of our Event Semantics are given in Figure 2. The Event 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, `click` or `focus` 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 Event Semantics in *handler registers*, $h \in \mathcal{H}$.

The Event Semantics, expectedly, needs to be aware of the configurations of the underlying language (L-configurations), $c \in \mathcal{C}$, but sees them as a black box and interacts with them only through an interface, presented shortly. It does assume that an L-configuration can be divided into: a *store component*, describing the variable store of L; and a *heap component*, describing the heap on which L-execution operates; and a *control flow component*, describing how the L-execution is to proceed. For example, a concrete JSIL configuration, $\langle \rho, \mu, m, cs, i\rangle$, consists of: a variable store ρ (the store component); a memory μ and a metadata table m (the heap 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). A symbolic JSIL configuration also includes a path condition, π, which is part of the control flow component. An L-configuration is *final iff* it cannot be executed further in the L-semantics. 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, $p \in \mathcal{P}$.

The L-semantics communicates with the Event Semantics via *event labels*, $\ell \in \mathcal{L}$, which represent the fundamental operations (primitives) through which we capture the behaviour of our targeted APIs. In particular, 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 labels are used in the modelling of DOM UI Events (cf. 3.2). Additionally, we support asynchronous computation scheduling via the schedule label, required for JS promises (cf. 4.1), and an asynchronous wait via the await label, required for JS `await` (cf. 4.2).

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 Event Semantics is to proceed. We consider two types of continuations:

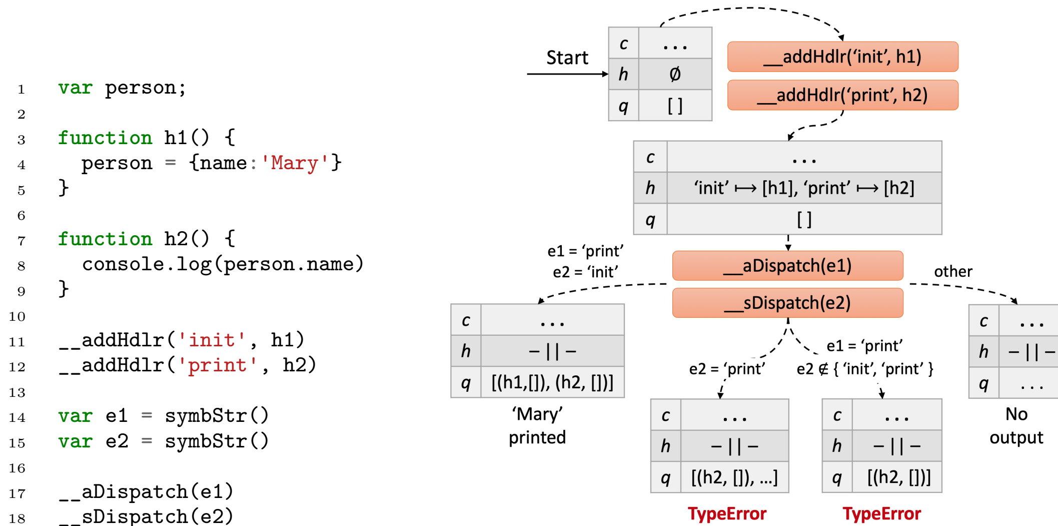
handler-continuations and yield-continuations. A *handler-continuation* is a pair, (f, v), essentially stating that the handler f is be to be executed with argument v. When an event is dispatched via sDispatch or aDispatch, the respective handler-continuations are put in the handler queue. A *yield-continuation* is a pair, (c, p), stating that the L-configuration c has been suspended and can be re-activated once the predicate p holds.

Finally, the Event Semantics configurations, (E-configurations), $\omega \in \Omega$, consist of: an L-configuration; a handler register; and a continuation queue.

Using the Event Semantics in JavaScript. Our JS reference implementations of the event-related APIs interact with the Event Semantics via *JS wrapper functions*, one per event label; we denote, for example, the wrapper function of the addHdlr label by `___addHdlr`, and the others analogously. Calls to these wrapper functions are meant to be intercepted by the underlying JavaScript implementation, which is then supposed to construct the corresponding label and pass it on to the Event Semantics. In JaVerT.Click, these wrapper functions resolve to JSIL functions with dedicated identifiers, the calls to which are then intercepted appropriately by the JSIL semantics. This approach, however, is independent of JaVerT.Click: any other implementation of JavaScript and of our Event Semantics can re-use our reference implementations, as long as these wrapper functions are properly intercepted.

Example. Below, we give a simple JavaScript example of how our Event Semantics can be used in JaVerT.Click (left), together with parts of its execution trace (right). In the E-configurations shown in the trace, we focus on the handler register and continuation queue, both of which are initially empty, and omit the details of the JSIL-configuration c.

First, in lines 1-9, we declare a variable `person` and two functions: `h1`, which initialises `person`, and `h2`, which prints out its `name`. Next, in lines 11-12, we add `h1` and `h2` as handlers for the `'init'` and `'print'` events, respectively, by using the wrapper function `___addHdlr`, exposed globally by the Event Semantics. This is recorded appropriately in the handler register, which then does not change for the remainder of the execution (denoted by $-||-$ in the diagram). Next, in lines 14-15, we declare `e1` and `e2` to be two symbolic events (strings), using the `symbStr`() function of JaVerT.Click. Finally, we dispatch `e1` asynchronously (line 17) and `e2` synchronously (line 18), using the appropriate wrapper functions. Intuitively,



in an asynchronous dispatch, the related handlers (here, any handlers for `e1`) are added to the *back* of the current continuation queue (here, an empty continuation queue), to be executed after all of the previously scheduled continuations are completed. In contrast, in a synchronous dispatch, the current computation is suspended and the related handlers (here, any handlers for `e2`) are added to the *front* of the continuation queue (which now contains the handlers for `e1`), to be executed immediately, followed by the remainder of the suspended current computation (which is empty, as the synchronous dispatch is at the end of the program, and is thus omitted from the diagram).

Given that the events are symbolic, the two dispatches will cause the execution of JaVerT.Click to branch; there are four relevant cases, as illustrated in the diagram. First, if `e1` equals `'print'` and `e2` equals `'init'`, the continuation queue after the dispatches will contain `h1` followed by `h2`, meaning that the execution will terminate successfully and `'Mary'` will be printed to the console. However, if `e2` equals `'print'` (meaning that `h2` will be put in the front of the continuation queue by the synchronous dispatch) or if `e1` equals `'print'` and `e2` has no associated handlers (meaning that `h2` will be put in the back of the continuation queue by the asynchronous dispatch, but will be the only function in that queue), the execution will throw a native JavaScript type error, as `h2` will attempt to read the `'name'` property of `person`, which will not have been initialised. Finally, in all other cases, the execution will terminate successfully, but with no output to the console.

Parametricity of the Event Semantics. As illustrated in Figure 1, the Event Semantics is implemented parametrically, as a layer on top of a given L-semantics. Since a unified presentation that reflects the implementation precisely would take up considerable space, we choose to present the concrete (§2.2) and the symbolic (§2.3) Event Semantics separately.

2.2 Concrete Event Semantics

A concrete Event Semantics is built on top of a concrete L-semantics. It interacts with L-configurations via an interface that consists of six functions: assume, $\mathsf{suspend}$, $\mathsf{initialConf}$, $\mathsf{isFinal}$, $\mathsf{mergeConfs}$, and $\mathsf{splitReturn}$; we describe these functions abstractly on their first use, and illustrate how some of them work in JSIL. The Event Semantics also uses the following auxiliary relations: **(1)** *add handler*, $\mathcal{AH}(h, e, f)$, for extending the handler register h with the handler f for an event e; **(2)** *remove handler*, $\mathcal{RH}(h, e, f)$, for removing the handler f for e from h; **(3)** *find handlers*, $\mathcal{FH}(h, e)$, for obtaining the handlers associated with e in h; and **(4)** *continue with*, $\mathcal{CW}_\mathrm{L}(c, \kappa)$, for updating the L-configuration c so that the continuation κ can be executed. We first give the formal definitions of these auxiliary relations, using function notation as they are deterministic in the concrete case. 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.

Concrete Event Semantics: Auxiliary Relations

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.
$$\mathcal{CW}_\mathrm{L}(c, (f, v)) \triangleq \mathrm{L}.\mathsf{initialConf}(c, (f, v))$$

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

These definitions are all straightforward except $\mathcal{CW}_\mathrm{L}$.When given a handler-continuation, $\kappa = (f, v)$, $\mathcal{CW}_\mathrm{L}$ sets up the execution of the handler f with argument v by using the $\mathsf{initialConf}$ function of the L-semantics interface, which returns an the L-configuration consisting of the the heap component of c and the control flow and store components set up to execute only the function f with argument v. When given a yield-continuation, $\kappa = (c', p)$, $\mathcal{CW}_\mathrm{L}$ requires the predicate p to hold for the current L-configuration c, in which case it merges the two configurations using the $\mathsf{mergeConfs}(c, c')$ function of the L-semantics interface, which returns a configuration that consists of the heap component of c and the control flow and store components of c'; in particular, in JSIL, given $c = \langle \rho, \mu, m, cs, i\rangle$ and $c' = \langle \rho', \mu', m', cs', i'\rangle$, we would have that $\mathsf{mergeConfs}(c, c') = \langle \rho', \mu, m, cs', i'\rangle$.

We now give the concrete Event Semantics transitions, which are of the form $\omega \rightsquigarrow^{\alpha}_{\mathsf{E}(\mathrm{L})} \omega'$, where ω and ω', respectively, are the configurations before and after the computed step, and α 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 $\alpha ::= \cdot \mid (e, v)$, where $\cdot$ represents no environment action and (e, v) represents the triggering of the event e with value v. For clarity, we elide $\cdot$ in the transitions.

Concrete Event Semantics: $\langle c, h, q\rangle \rightsquigarrow^{\alpha}_{\mathsf{E}(\mathrm{L})} \langle c', h', q'\rangle$

Language Transition
$$\frac{c \rightsquigarrow_\mathrm{L} c'}{\langle c, h, q\rangle \rightsquigarrow_{\mathsf{E}(\mathrm{L})} \langle c', h, q\rangle}$$

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

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

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

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

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

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

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

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

Continuation-Failure
$$\frac{\mathrm{L}.\mathsf{isFinal}(c) \quad q = \kappa : q' \quad (c, \kappa) \notin \mathsf{dom}(\mathcal{CW}_\mathrm{L})}{\langle c, h, q\rangle \rightsquigarrow_{\mathsf{E}(\mathrm{L})} \langle c, h, q' +\!\!+ [\kappa]\rangle}$$

The first seven rules rely on a transition of the L-semantics, updating the current L-configuration with the one generated by the L-transition and using the generated label to determine which event-related action is to be performed, if any. The first three rules are straightforward; we describe the remaining four below:

[Synchronous Dispatch] When the L-semantics generates the label $\mathsf{sDispatch}\langle e, v\rangle$, the Event Semantics first creates a handler-continuation for each handler associated with e, together with a yield continuation, $(c', (\lambda c.\mathsf{True}))$. 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 Event Semantics uses the $\mathsf{suspend}(c')$ function of the L-semantics, which returns the configuration that is the same as c' but marked as final, to construct a final configuration c'', which, given [Continuation-Success], means that the execution of c' will stop and the first handler will be executed next.

[Asynchronous Dispatch] When the L-semantics generates the label $\mathsf{aDispatch}\langle e, v\rangle$, the Event 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 label $\mathsf{schedule}\langle f, v\rangle$; the Event Semantics creates a handler-continuation (f, v) 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 label $\mathsf{await}\langle v, p\rangle$, the Event Semantics creates the return configuration, c_r, and the await configuration, c_a via the splitReturn function of the L-semantics interface, which constructs: c_r from c by setting up the control flow component as if the currently executing function, f, returned the value v; and c_a from c 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 p 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 label (e, v), and the Event 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 $\mathsf{isFinal}(c)$ function of the L-semantics interface, which returns true if c is final, and false otherwise), the Event 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]).

2.3 Symbolic Event Semantics

Symbolic execution [2, 3, 4] 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 Event Semantics introduced in §2.2, obtained by lifting the concrete event semantics to the symbolic level, following well-established approaches [44, 43, 12].

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 introduced in §2.1 are defined as in Figure 2, but for symbolic instead of concrete values, and are annotated with ˆ to be distinguishable from their concrete counterparts; e.g., we have: *symbolic events*, $\hat{e} \in \widehat{\mathcal{E}} \subset \widehat{\mathcal{V}}$; *symbolic handler registers*, $\hat{h} \in \widehat{\mathcal{H}}$, mapping symbolic events to lists of function identifiers; and symbolic configurations, $\hat{\omega} \in \widehat{\Omega}$, comprising a symbolic L-configuration, $\hat{c} \in \widehat{\mathcal{C}f}$, a *symbolic handler register*, and a *symbolic continuation queue*, $\hat{q} \in \widehat{\mathcal{Q}}$. We also assume that every symbolic L-configuration $\hat{c}$ contains a boolean symbolic value, $\pi \in \Pi \subset \widehat{\mathcal{V}}$, to which we refer as the *path condition* of $\hat{c}$.

The symbolic Event Semantics, like the concrete, uses the L-semantics interface and the four auxiliary relations introduced in §2.2. When executed symbolically, however, the auxiliary relations that operate on handler registers ($\mathcal{AH}$, $\mathcal{RH}$, and $\mathcal{FH}$) may branch. To account for this branching, we pair each outcome with a constraint describing the conditions under which the outcome is valid. The formal definitions are given below; we omit the definition of the $\mathcal{RH}$ relation, as it is analogous to that of $\mathcal{AH}$.

Symbolic Event Semantics: Auxiliary Relations

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

Add Handler - Not Found
$$\frac{\hat{h}' = \hat{h}\,[\hat{e} \mapsto [f]]}{\mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \hat{e} \notin \texttt{dom}(\hat{h}))}$$

Find Handler - Found
$$\frac{\hat{e}' \in \texttt{dom}(\hat{h})}{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow (\hat{h}(\hat{e}'), \hat{e} = \hat{e}')}$$

Find Handler - Not Found
$$\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([], \hat{e} \notin \texttt{dom}(\hat{h}))$$

An excerpt of the symbolic Event Semantics is given below. We focus on the representative rules 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 $\mathsf{assume}(\hat{c}, \pi)$ function of the L-semantics interface, which returns the symbolic L-configuration obtained by extending the path condition of $\hat{c}$ with the formula π if such an extension is satisfiable, and is undefined otherwise.

Symbolic Event Semantics (excerpt): $\langle \hat{c}, \hat{h}, \hat{q}\rangle \rightsquigarrow^{\hat{\alpha}}_{\hat{\mathsf{E}}(\mathrm{L})} \langle \hat{c}', \hat{h}', \hat{q}'\rangle$

Add Handler
$$\frac{\hat{c} \rightsquigarrow^{\hat{\ell}}_{\mathrm{L}} \hat{c}' \quad \hat{\ell} = \mathsf{addHdlr}\langle \hat{e}, f\rangle \\ \mathcal{AH}(\hat{h}, \hat{e}, f) \rightsquigarrow (\hat{h}', \pi) \quad \hat{c}'' = \mathrm{L}.\mathsf{assume}(\hat{c}', \pi)}{\langle \hat{c}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}(\mathrm{L})} \langle \hat{c}'', \hat{h}', \hat{q}\rangle}$$

Environment Dispatch
$$\frac{\mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, [\hat{e}, \hat{v}]) \mid_{i=0}^n] \\ \hat{c}' = \mathrm{L}.\mathsf{assume}(\hat{c}, \pi)}{\langle \hat{c}, \hat{h}, \hat{q}\rangle \rightsquigarrow^{(\hat{e}, \hat{v})}_{\hat{\mathsf{E}}(\mathrm{L})} \langle \hat{c}', \hat{h}, \hat{q} +\!\!+ \hat{q}'\rangle}$$

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

Asynchronous Dispatch
$$\frac{\hat{c} \rightsquigarrow^{\hat{\ell}}_{\mathrm{L}} \hat{c}' \quad \hat{\ell} = \mathsf{aDispatch}\langle \hat{e}, \hat{v}\rangle \\ \mathcal{FH}(\hat{h}, \hat{e}) \rightsquigarrow ([f_i \mid_0^n], \pi) \quad \hat{q}' = [(f_i, [\hat{e}, \hat{v}]) \mid_{i=0}^n] \\ \hat{c}'' = \mathrm{L}.\mathsf{assume}(\hat{c}', \pi)}{\langle \hat{c}, \hat{h}, \hat{q}\rangle \rightsquigarrow_{\hat{\mathsf{E}}(\mathrm{L})} \langle \hat{c}'', \hat{h}, \hat{q} +\!\!+ \hat{q}'\rangle}$$

Correctness. To establish the correctness of the symbolic Event Semantics w.r.t the concrete Event Semantics, we first relate the corresponding 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 2 component-wise, overloading notation: for example, $\mathcal{I}_\varepsilon(\langle \hat{c}, \hat{h}, \hat{q}\rangle) \triangleq \langle \mathcal{I}_\varepsilon(\hat{c}), \mathcal{I}_\varepsilon(\hat{h}), \mathcal{I}_\varepsilon(\hat{q})\rangle$. We assume that interpretation is preserved by the functions of the L-semantics interface; for example, that $\mathrm{L}.\mathsf{isFinal}(\hat{c}) \Leftrightarrow \mathrm{L}.\mathsf{isFinal}(\mathcal{I}_\varepsilon(\hat{c}))$.

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

The correctness of the Event Semantics relies on the correctness of the L-semantics. A given symbolic L-semantics is correct w.r.t. a given concrete L-semantics, as formalised in Definition 1, 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*). Directed Completeness, in particular, guarantees the absence of false-positive bug-reports: if a bug happens symbolically, then it must also happen concretely.

▶ **Definition 1** (Correctness Criteria - Symbolic L-Semantics).

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

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

Theorem 2 states that if the symbolic L-semantics is correct, then so is the obtained Event Semantics. To precisely identify the concrete traces that follow the same path as the symbolic trace, in Theorem 2 we only pick concretisations of the initial symbolic state that satisfy the *final* path condition ($\pi = \mathsf{pc}(\hat{\omega}')$).

▶ **Theorem 2** (Correctness of the Symbolic Event Semantics).

E-Directed-Soundness
$$\hat{\omega} \rightsquigarrow^{\hat{\alpha}}_{\hat{E}(\mathrm{L})} \hat{\omega}' \wedge \pi = \mathsf{pc}(\hat{\omega}') \wedge (\varepsilon, \omega) \in \mathcal{M}_\pi(\hat{\omega}) \wedge (\varepsilon, \alpha) \in \mathcal{M}_\pi(\hat{\alpha}) \wedge \omega \rightsquigarrow^{\alpha}_{E(\mathrm{L})} \omega' \implies (\varepsilon, \omega') \in \mathcal{M}_\pi(\hat{\omega}')$$

E-Directed-Completeness
$$\hat{\omega} \rightsquigarrow^{\hat{\alpha}}_{\hat{E}(\mathrm{L})} \hat{\omega}' \wedge \pi = \mathsf{pc}(\hat{\omega}') \wedge (\varepsilon, \omega) \in \mathcal{M}_\pi(\hat{\omega}) \implies \exists \alpha, \omega'.\ \omega \rightsquigarrow^{\alpha}_{E(\mathrm{L})} \omega'$$

We actually prove a stronger result, analogous to that given in Definition 1, with $\pi \Rightarrow \mathsf{pc}(\hat{\omega}')$, from which the presented result trivially follows. The proof is done by case analysis on the symbolic rules for the Event Semantics, and can be found integrally in [31].

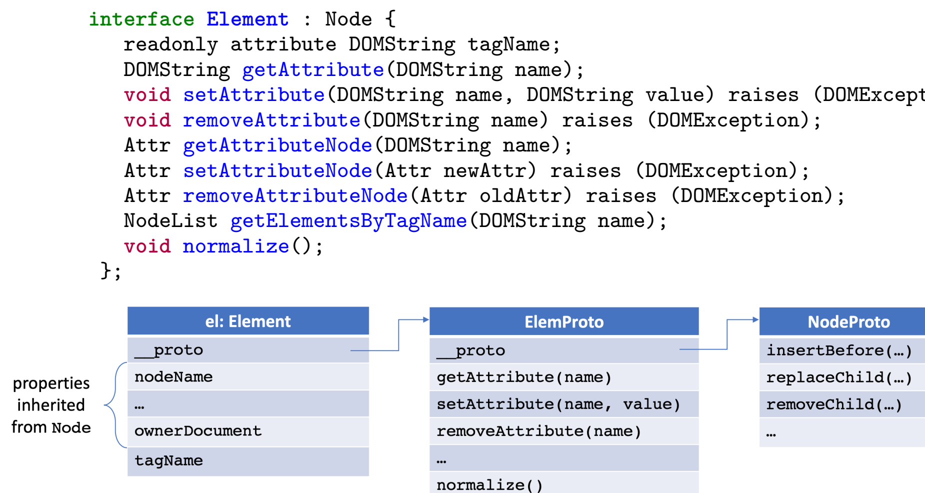
3 The DOM API

The Document Object Model (DOM) [53] is an API through which the code executing in the browser can interact with the Web page 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. To cope with this growing complexity, the DOM API has been organised as a collection of smaller APIs, each targeting a specific set of features. Recently, the most relevant of these APIs, Core Levels 1-3 [47, 49], have been unified in a single all-encompassing DOM API, called the DOM Living Standard [53], which defines a "platform-neutral model for events, aborting activities, and node-trees". The DOM Living Standard is inspired by the ECMAScript standard [7]. It is written as if it were the pseudo-code of a DOM implementation, describing each DOM method *operationally* and detailing each evaluation step. This approach, unlike the previous declarative one [47, 48, 49], facilitates new reference implementations tightly connected to the text of the standard.

In this section, we present our JavaScript reference implementations of two DOM APIs: DOM Core Level 1 [47], which describes a range of operations for inspecting and updating XML/HTML documents (§3.1); and DOM UI Events [53], which describes the event model of the DOM (§3.2). For the latter, we describe in detail its connection to the Event Semantics. Importantly, both reference implementations are *trustworthy*: they closely follow the specifications of their corresponding methods as per the DOM Living Standard, as illustrated in this section; and they were thoroughly tested against the appropriate official test suites, as shown in §5. They, therefore, constitute a reliable representation of the DOM, which is useful for analysing Web programs that interact with the DOM API.

3.1 DOM Core Level 1

The DOM Core Level 1 API [47] 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



```
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 3 DOM Element interface (top) and the respective JavaScript object graph (bottom).

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`.

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; for instance, every `Element` node is also a `Node`, meaning that it exposes all fields and methods defined in the `Node` interface.

We implement the DOM Core Level 1 API 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 the following, we describe our implementations of the `Element` and `NodeList` interfaces, which showcase, respectively, how our implementation follows the standard, and how JavaScript enables us to write an elegant implementation of DOM live collections.

Element Interface. In Figure 3, we show the `Element` interface written in IDL (Interface Description Language) as in the standard (top) and a fragment of its corresponding object graph from our JavaScript implementation (bottom). The standard states that `Element` inherits from `Node`, meaning that all objects of type `Element` expose the methods and fields of `Node` objects. Additionally, every `Element` object exposes a field `tagName` and the methods `getAttribute`, `setAttribute`, `removeAttribute`, `getAttributeNode`, `setAttributeNode`, `removeAttributeNode`, `getElementsByTagName`, and `normalize`.

In the JavaScript object graph, 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`.

NodeList Interface. 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.

3.2 DOM UI Events

The DOM UI Events API [53] 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.

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 ([53], §2.9). 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.

In the following, we describe our implementation of the DOM `Dispatch`, demonstrate that this implementation follows the pseudo-code of the standard line-by-line, and describe in detail how it is connected to the Event Semantics.

DOM Dispatch. We explain the DOM `Dispatch` algorithm via an example given in Figure 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 *capture phase*, where the

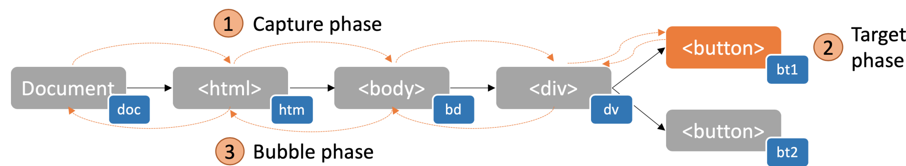


Figure 4 DOM Dispatch Phases.

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, useCapture)`, allows the programmer to specify if a given handler is to be executed in the capture phase or the bubble phase through the `useCapture` boolean; 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 execution alter the propagation path, those changes will not be taken into account by the `Dispatch` algorithm.

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

```
1  function Dispatch(event, target, flags) {
2    var relatedTarget = retarget(event.relatedTarget, target);
3    var touchTargets = getTouchTargets(event, target);
4    var actTarget = isActivationTarget(event);
5    updatePropagationPath(event, target, relatedTarget, touchTargets, actTarget);
6    captureAndTarget(event, flags)
7    if (event.bubbles) { bubble(event, flags) }
8    clear(event);
9    return !event.canceled
10 }
```

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. 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.
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`.

Using the Event Semantics. In related works [19, 29], 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 Event 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 Event 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 Event Semantics. It is then the job of this function, rather than the Event Semantics, to traverse the DOM tree, starting at the node where the event was triggered, and execute the user-register handlers in the appropriate order.

Below, we show our implementation of the `dispatchEvent` function, used to model programmatic dispatch of DOM events. This function calls the Event Semantics synchronous dispatch wrapper, `__sDispatch`, in line 5. The behaviour of the sDispatch primitive, as given in §2, precisely captures the programmatic DOM event dispatch as per the standard, where the associated event handlers are meant to be executed immediately.

```
1  function dispatchEvent(event, flags) {
2    if (event.dispatch || !event.initialized) {
3      throw new DOMException(INVALID_STATE_ERR) };
4    event.isTrusted = false; event.target = this;
5    return __sDispatch(event, this, flags)
6  }
```

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 below. 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.

```
1  function innerInvoke (event, listeners, phase, legacyOutputDidListenersThrowFlag) {
2    var found = false;                           // 1. Let found be false.
3    for (var i = 0; i < listeners.length; i++) {  // 2. For each listener in
     ↪  listeners...
4      if (listener.removed) continue;              // ...whose removed is false:
5      // 2.1. If event's type attribute value is not listener's type, then continue.
6      if (event.type !== listener.type) continue;
7      // 2.2. Set found to true.
8      found = true;
9      // 2.3. If phase is "capturing" and listener's capture is false, then continue.
10     if ((phase === "capturing") && (listener.capture === false)) continue;
11     // 2.4. If phase is "bubbling" and listener's capture is true, then continue.
12     if ((phase === "bubbling") && (listener.capture === true)) continue;
13     // 2.5. If listener's once is true, then remove listener from event's
       ↪  currentTarget attribute value's event listener list.
14     if (listener.once === true) event.currentTarget.removeListener(listener);
```

```
15      ...
16      // 2.10. Call a user object's operation with listener's callback, "handleEvent",
        ↪  event, and event's currentTarget attribute value.
17      execCallBack(listener.handleEvent, "handleEvent", event, event.currentTarget);
18      ...
19      // 2.13. If event's stop immediate propagation flag is set, then return found.
20      if (event.stopImmediatePropagation === true) return found;
21    }
22    return found; // 3. Return found
23  }
```

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. This behaviour is actually disallowed by the DOM standard, which defines the `bubbles` attribute as read-only, but is exhibited by the DOM engines of Chrome, Edge, Firefox, and Safari. Our reference implementation does not suffer from this problem as we define read-only attributes as non-writable on creation.

4 JavaScript Promises and async/await

Promises were introduced into JavaScript (JS) in the 6th version of the standard [8], 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 [14], which is extremely difficult to understand and reason about.

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*.

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.

This section describes our reference implementations of the JS Promises and `async/await` APIs, as described in sections 25.6, 25.7, and 6.2.3.1 of the 9th version of the ECMAScript standard [9]. Analogously to the DOM reference implementations, these APIs: are implemented directly in JS (ES5 Strict), with the Promises implementation following the standard line-by-line; are thoroughly tested against the latest version of the official ECMAScript test suite [6] (cf. §5); and make use of their dedicated Event Semantics primitives (cf. §2).

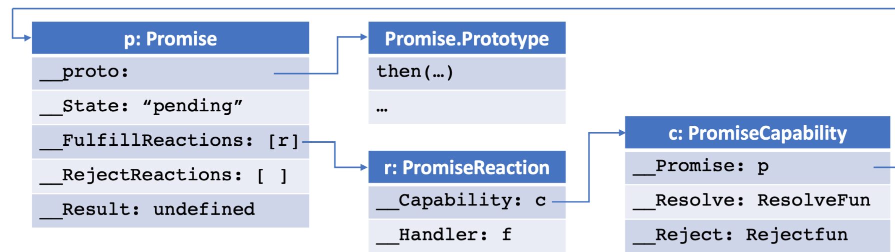


Figure 5 Promises Object Graph.

4.1 Promises API

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. Consider the following example:

```
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)
```

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.

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*, whose definitions in the ECMAScript standard are also operational, intricate, and intertwined.

The structure of Promise objects is also fairly complex. We illustrate this by giving, in Figure 5, the object graph associated with the promise `p` of the example 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 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.

Using the Event Semantics. Our reference implementation of JS promises interacts with the Event 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 wrapper function of the schedule primitive of the Event 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      __schedule(reactionJob);
6    }
7  }
```

Note that the Event Semantics schedule primitive, as defined in §2, 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 [9], 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, 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 5), 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.

```
1   function FulfillPromise(promise, value) {
2     // 1. Assert: The value of promise's [[State]] internal slot is "pending".
3     Assert(promise.__State === "pending");
4     // 2. Let reactions be the value of promise's [[FulfillReactions]] internal slot.
5     var reactions = promise.__FulfillReactions;
6     // 3. Set the value of promise's [[Result]] internal slot to value.
7     promise.__Result = value;
8     // 4. Set the value of promise's [[FulfillReactions]] internal slot to undefined.
9     promise.__FulfillReactions = undefined;
10    // 5. Set the value of promise's [[RejectReactions]] internal slot to undefined.
11    promise.__RejectReactions = undefined;
12    // 6. Set the value of promise's [[State]] internal slot to "fulfilled".
13    promise.__State = "fulfilled";
14    // 7. Return TriggerPromiseReactions(reactions, value).
15    return TriggerPromiseReactions (reactions, value)
16  }
```

4.2 async/await

The `async/await` APIs are defined in Sections 6.3.1 and 25.7 of the 9th version of the ECMAScript standard [9]; they are 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.

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:

```
1 async function f () { if (b === true) { return 1 } else { throw 2 } };
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 program below:

```
1 var p = new Promise(function (resolve, reject) { ... });
2 async function g () { return await p };
3 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.

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.

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 translation[1] of asynchronous functions:

$$\begin{aligned}\mathcal{C}\langle \text{async function}(\bar{x})\{s\}\rangle \triangleq\ & \text{function}(\bar{x})\ \{\\ & \quad \text{return new Promise(function(resolve, reject)}\ \{\\ & \quad\quad \text{try}\ \{\mathcal{C}_a\langle s\rangle;\ \text{resolve(undefined)}\}\ \text{catch(e)}\ \{\text{reject(e)}\}\\ & \quad \})\\ & \}\end{aligned}$$

[1] 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.

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`. Hence, the function `f`, given in Code Snippet 1, is compiled to:

```
1  function f () {
2    return new Promise (function (resolve, reject) {
3        try { if (b === true) { resolve(1); return } else { throw 2 } }
4          catch (x) { reject(x); return }
5      })
6  }
```

The compilation of the `await p` expression is more involved. Concretely, the compiled code calls the wrapper function for the Event Semantics `await` primitive with the argument `getPredicate(p)`, which corresponds to a function that evaluates to `true` once the promise `p` has been settled. Given the definition of `await` in §2, this precisely corresponds to the core behaviour of the JS `await`. 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. Below, we illustrate the compilation of the function `g`, given in Code Snippet 2.

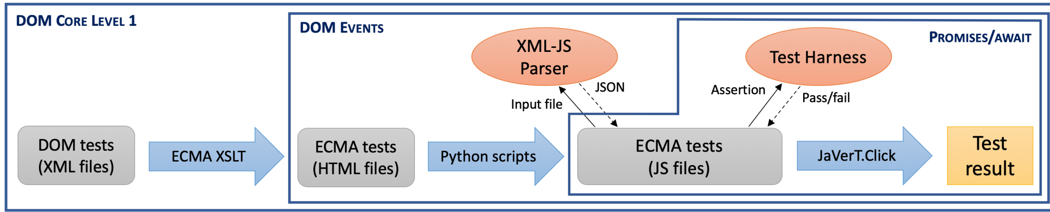
```
1  function g () {
2    return new Promise (function (resolve, reject) {
3       try {
4          __await(getPredicate(p));
5          if(p.__State === "resolved") { resolve(p.__Result) } else { throw p.__Result }
6       } catch (x) { reject(x); return }
7     })
8  }
```

5 Evaluation

We show that our reference implementations of DOM Core Level 1, DOM UI Events, JS Promises, and `async`/`await` are trustworthy by passing all applicable tests from their official test suites [46, 50, 6] in JaVerT.Click. In doing so, we discover coverage gaps in the DOM Core Level 1 and UI Events test suites and create additional tests to complete their coverage. We demonstrate that JaVerT.Click can reason about real-world JS code by creating a comprehensive symbolic test suite for the `events` module of the `cash` library [55], a widely-used alternative for jQuery. Our symbolic testing establishes bounded correctness of several essential properties of the library and reveals two subtle, previously unknown bugs. We also symbolically test the `p-map` library [35], which adds an extra layer of functionality on top of JS promises. We achieve 100% line coverage and discover one bug. All three bugs discovered by JaVerT.Click in `cash` and `p-map` have been reported and have since been fixed.

5.1 Testing the Reference Implementations

To run the test suites, we establish a common testing infrastructure, illustrated below. The tests for Promises and `async`/`await` are written in JS. To run them in JaVerT.Click, we only need to compile the ECMAScript test harness together with the tests. The tests for DOM Events, in contrast, are written in HTML and contain JS scripts enclosed by the `<script>` tag. Using Python scripts, we first isolate this code into a JS test file, then add to it the JSON object obtained from the appropriate input XML file using the `xml-js` parser [58]. Finally, as the DOM Core Level 1 tests are written in XML, we additionally have to first transform them into HTML tests using XSLT.



We present the results of testing our reference implementations against their appropriate official test suites. For each implementation, we provide the number of: available tests in the test suite; applicable tests; and passing tests. Additionally, for three of the test suites, we give: its computed line coverage; the number of untested lines in its standard; and the number of additional tests that we created to complete its coverage. Given that we pass all of the applicable tests, which have substantial coverage of their respective standards, for all four APIs, we have strong confidence that our reference implementations are indeed correct. Interestingly, the test suite repository for DOM Events also provides the testing results for four browsers: Chrome, Edge, Firefox and Safari [57]. These results show that no single browser fully passes the test suite, and that, out of the 56 tests that we pass, 12 fail in at least one of the four browsers.

	Core Level 1	**Events**	**Promises**	`async/await`
Available Tests	527	83	474	86
Applicable Tests	527	56	344	68
Passing Tests	**527**	**56**	**344**	**68**
Test Suite Line Coverage	98.14%	97.45%	98.76%	N/A
Number of Untested Lines	13	8	5	N/A
Additional Tests	5	3	N/A	N/A

For three of the test suites, some tests need to be filtered out due to the coverage of either our reference implementations or JaVerT.Click. For DOM Events, we filter out 1 test that uses ES6 classes, 5 that use the postMessage API, 2 that use the AJAX API, and 19 that use unsupported CSS features (scrolling and animation). For Promises, most filtered tests (106/130) are due to ES6 Symbol iteration; once we support this feature, these tests should pass as similar tests that use Array iteration already pass. We also filter out 21 tests that use other unsupported ES6 features (classes, reflection, and proxies), and 3 that require non-strict mode. For `async/await`, we filter out 14 tests that use ES6 default arguments and 4 that use ES6 generators.

When it comes to test suite coverage, we observe that it is comprehensive, but incomplete. We have inspected the filtered tests for DOM Events and JS Promises and believe that they would not trigger the missing lines. Note that we are not able to perform a proper coverage analysis for `async/await`, as we do not follow its description in the standard line-by-line.

Observations. We have found the ECMAScript standard to be written and tested with a higher degree of rigour than the DOM Living Standard. It is self-contained and precise, with no implicit assumptions and discrepancies between the standard and the test suite.

The DOM Living Standard, in contrast, uses features from other standards, such as HTML and the Shadow DOM [26], without providing any intuition. This meant that we needed to understand multiple standards written in different formats and had to read

substantial additional documentation (e.g., the Mozilla Web Docs [25]) in order to model the API behaviour correctly. Additionally, the DOM Living Standard interfaces do not have a well-defined scope. For instance, the standard makes clear that every `EventTarget` object has an associated event listener list, but this list is not declared as an attribute of the `EventTarget` interface. This can lead to different interpretations by implementors. Finally, we found a few discrepancies between the DOM Standard and the official test suites that are likely to cause difficulties for implementors: for example:

- In DOM Core Level 1, on setting `Attr.value`, the standard only states that a `Text` node with the unparsed contents of the provided value should be created; the tests additionally require that this text node be inserted as a child node of the attribute.
- In DOM Events, for `Event.isTrusted`, the standard defines the `isTrusted` property of the `Event` interface to be a boolean that is used to indicate whether or not the `dispatchEvent` function was used; the tests specifically require the `isTrusted` property to be an accessor property and to have a dedicated getter.

5.2 Symbolic Testing of the `cash` Library

The `cash` library [55] is a jQuery 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. It has a growing community of users, with more than 10K weekly downloads and 735K overall downloads on npm [56], and more than 4.4K stars on GitHub [55].

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. This module has five main and twelve auxiliary functions. Here, we focus on the main functions, presented below:

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

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

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

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

.trigger: `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 coverage of `.on` is 76.92%, of `.trigger` is 93.75%, of `.ready` is 0% and of the main auxiliary function of `.on` is 81.82%; the remaining functions have 100% coverage. We complete the coverage of the concrete test suite for the `events` module by writing five additional concrete tests.

5.2.1 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 1, we give, for these tests, their execution time (Time, in seconds) and the number of executed JSIL commands (JSIL 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 has been registered to a given event using `.on`, 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 §5.2.2.

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

tOff : If a handler registered to an event is deregistered using `.off`, 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 as the length of the longest property of the JavaScript initial heap, `propertyIsEnumerable`. It 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.

5.2.2 The 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 event model of the browser. However, the semantics of events in `cash` differs from that of the browser 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 `String.prototype.split`. This implementation, however, causes two subtle bugs, discovered by the `sHand` test, whose stylised code, with detailed inlined explanations, is given below:

```
1  var count = 0, ele = $('.event'); // Initialise counter and target element
2
3  function h () { count++ } // Handler counts the number of times it was called
```

Table 1 Symbolic Test Suite for the `events` module of `cash`.

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

```
4
5  // Create two symbolic events, e1 and e2, of maximum length 20
6  var e1 = symbStr(20), e2 = symbStr(20);
7
8  // Register the handler for e1 on ele, then trigger e2 on ele
9  ele.on(e1, h); ele.trigger(e2);
10
11 Assert(
12   // Handler was executed only once, if e1 and e2 were equal and non-empty,
13   (count === 1 && e1 === e2 && e1 !== "") ||
14   // and was not executed otherwise.
15   (count === 0 && (e1 !== e2 || e1 === ""))
16 );
```

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 [32]: 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` 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 is not executed even though `e1` and `e2` are equal, whereas, for the second two, it is executed despite `e1` and `e2` being different. This bug is also caused by the redirection done in the `getEventTypeBubbling` function. Precisely, this redirection is applied in the `.on`, `.one`, and `.off` functions, but not in the `.trigger` function, effectively meaning that 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.

Both bugs have been reported to the developers of `cash`,[2] and have since been fixed. The first bug also exists in jQuery, where it will be corrected for the upcoming 4.0 version.[3]

5.3 Symbolic Testing of the `p−map` Library

The `p−map` library [35] is a small JavaScript library that extends the functionality of JavaScript promises with the ability to concurrently map over pending promises. It has more than 10M weekly downloads and 825M overall downloads on npm [36], and 532 stars on

[2] https://github.com/kenwheeler/cash/issues/317, 318
[3] https://github.com/jquery/jquery/issues/3256

GitHub [35]. It calls both the JavaScript Promises and JavaScript `async/await` APIs. We performed symbolic testing of `p−map`, where we achieved 100% line coverage and discovered a bug that allowed the number of concurrently handled promises to go above its declared maximum due to the library using non-integer numbers. This bug has been reported to and fixed by the developers of `p−map`.[4] For space reasons, we delay the full account of our analysis of `p−map` to a future publication.

6 Related Work

We believe we are the first to provide a general infrastructure for symbolic analysis of modern event-driven Web applications. There has been prior work on formalising and analysing specific event-driven Web APIs, such as DOM UI Events [29, 19] and JavaScript Promises [22, 1], as well as Node.js events [21, 23]. However, to the best of our knowledge, there is no prior work on formalising the JavaScript async/await API. Hence, we focus the discussion on: **(1)** axiomatic and operational semantics for DOM Core Level 1; **(2)** operational semantics for DOM Events; **(3)** operational semantics for JavaScript Promises; and **(4)** symbolic execution for JavaScript programs that interact with the DOM.

Axiomatic/Operational Semantics of DOM Core Level 1. Based on context logic [5], Smith et al. introduced an axiomatic semantics [15] for a small fragment of DOM Core Level 1, proving it sound with respect to their operational semantics. In his PhD thesis [34], Smith 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 on the extended interfaces. 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.

Several operational semantics for different fragments and adaptations of DOM Core Level 1 were proposed for various types of analysis, such as information flow control [24, 30], type systems [42] and abstract interpretation [18], 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 525 tests of the official test suite [46]. 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.

Operational Semantics for DOM Events. In this context, the work closest to ours is [19], which presents the first operational model for reasoning about DOM events. This model consists of a Scheme [38] 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, however, is not tested against the official DOM Events test suite and does not have a line-by-line correspondence with the text of the standard.

[4] `https://github.com/sindresorhus/p-map/issues/26`

There are multiple tools for analysing event-driven JavaScript programs based on different types of program analyses, such as information flow control [29, 45], type systems [28], and abstract interpretation [27]. Of these tools, only [29] comes with a formal semantics of DOM events. Concretely, the authors propose a simplified DOM event semantics instrumented with a sound information-flow monitor, and implement the monitor instrumentation on top of Webkit [41], 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 56 tests of the official test suite (100% of the appropriate tests, given our current coverage).

A Core Semantics for JavaScript Promises. Madsen et al. [22] 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} [17], 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. Later, Alimadadi et al. [1] extend 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 [33] for finding and explaining promise-related bugs in JS code.

The λ_{JS}-calculus [17] was justified by a desugaring function from ES5 that has been tested against the official Test262 test suite [6]. In contrast, λ_p does not come with a desugaring function from ES6 to λ_p and hence has not been tested against the promises-related part of Test262. Whilst λ_p 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 the symbolic testing presented in the paper. For this reason, we took great care in justifying its correctness.

Symbolic Execution for the DOM. Symbolic reasoning about the DOM in the literature is mostly focussed on bug-finding and/or automatic concrete test generation. For example, ConFix [10] uses concolic execution to generate DOM fixtures that allow high-coverage testing of JavaScript functions that use the DOM; however, it does not support DOM events and dynamically generated code using `eval` that interacts with the DOM. V-DOM [59] creates test suites by analysing both server- and client-side code, but only considers handlers that were registered statically (via HTML code, e.g. `<button onclick="myFunction()"/>`) and does not support dynamic handler registration (via `addEventListener`()).

There are several tools focussed on finding dependencies between event handlers, such as SymJS [20] and JSDep [37], which then use this information to automatically generate tests in the form of event triggering sequences. SymJS identifies handler dependencies by performing a dynamic write-read analysis. 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. JSDep 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.

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 [16], which provides its own implementation of the DOM event dispatch algorithm) and none offer a justification with respect to their representation of the DOM. In contrast, we provide complete, 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 broad 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, the above-mentioned tools do generate their concrete test suites automatically, while in JaVerT.Click, the developers have to write symbolic tests themselves.

7 Conclusions and Future Work

We have introduced a Core Event Semantics that is simple in design, yet expressive enough to capture the essence of three fundamental, complex event-related APIs, namely DOM UI Events, JavaScript Promises and `async/await`. To accompany the Core Event Semantics, we have created reference implementations of these three APIs, as well as a reference implementation of DOM Core Level 1, which underpins DOM UI Events. Our reference implementations are trusted, in the sense that all except that of `async/await` follow their respective standards line-by-line, and all are thoroughly tested against their official test suites. Together, the Core Event Semantics and the reference implementations form a trusted infrastructure that enables symbolic analysis of modern event-driven Web programs.

We have demonstrated that our infrastructure can be used in practice by implementing the Core Event Semantics, closely following the theory, on top of JaVerT 2.0, a state-of-the-art tool for JavaScript symbolic analysis. We have used the resulting tool, JaVerT.Click, to symbolically test two real-world libraries: `cash` and `p−map`, both with with 100% line coverage, establishing bounded correctness of several important properties and discovering three bugs in the process. To our knowledge, this is the first time that reasoning about multiple event-based APIs is supported either in a single formalism or in a single tool.

As part of the overall testing process, we have additionally discovered coverage gaps in the official test suites of DOM Core Level 1 and DOM UI Events, as well as in the concrete test suite of `cash`, and have created appropriate concrete tests that address these gaps.

We plan to extend this work in several directions. First of all, following the methodology that we have introduced in this paper, we will add support for other event-based APIs, such as the File [51], postMessage [54], and the Web Workers APIs [52], to the Core Event Semantics and JaVerT.Click. For each new API, this amounts to providing a trusted reference implementation in JavaScript, and extending the Event Semantics with any new appropriate event primitives that may be required. For instance, supporting the Web Workers API will require the Event Semantics to be extended with basic message-passing facilities. Our over-arching goal is to create a minimal event model expressive enough to reason about all widely-used Web APIs natively supported by major browsers.

We also intend to analyse further real-world libraries that are clients of our supported APIs. For example, PreactJS [39], a fast and light alternative to ReactJS [40], appears to be an excellent first target, as it is relatively small yet very successful, and is already being used by several major industrial players.

Another avenue to explore, given our trusted infrastructure, would be how to extend the full verification facilities of JaVerT.Click in order to be able to prove both meta-properties of the APIs themselves as well as correctness properties of programs that interact with the DOM and/or use event-related APIs.

We will also implement the Event Semantics as a layer on top of Gillian [11], our new multi-language platform for compositional symbolic analysis, by instantiating the Event Semantics with Gillian's intermediate language, GIL. There, in addition to JS code, we plan to reason about WebAssembly and Rust code that interacts with various event-based APIs.

Finally, we plan to design a policy language that would allow the developers to specify the event sequences of interest, given the programs they would like to analyse. These policies might play a role in automatically generating tests, as in the discussed related work [20, 37], but also in the broader context of symbolic analysis, where they would limit the branching that arises from exploring all possible event sequences triggered by the environment.

References

1 S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *PACMPL*, 2(OOPSLA):162:1–162:26, 2018.
2 R. Baldoni, E. Coppa, D. Cono D'Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.
3 C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE*, 2011.
4 C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56:82–90, 2013.
5 C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.
6 ECMA TC39. Test262 Test Suite. `https://github.com/tc39/test262`, visited 05/2020.
7 ECMA TC39. The ECMAScript Standard. `https://www.ecma-international.org/publications/standards/Ecma-262.htm`, visited 05/2020.
8 ECMA TC39. The ECMAScript Standard - 6th Edition. `http://www.ecma-international.org/ecma-262/6.0/`, visited 05/2020.
9 ECMA TC39. The ECMAScript Standard - 9th Edition. `http://www.ecma-international.org/ecma-262/9.0/`, visited 05/2020.
10 A. M. Fard, A. Mesbah, and E. Wohlstadter. Generating Fixtures for JavaScript Unit Testing (T). In *ASE*, 2015.
11 J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and Philippa G. Gillian, Part 1: A Multi-language Framework for Symbolic Execution. In *PLDI*, 2020.
12 J. Fragoso Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner. Symbolic Execution for JavaScript. In *PPDP*, 2018.
13 J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *PACMPL*, 3(POPL):66, 2019.
14 K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript. In *ESEM*, 2015.
15 P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. Local Hoare Reasoning about DOM. In *PODS*, 2008.
16 Gargoyle Software Inc. HTMUnit. `http://htmlunit.sourceforge.io/`, visited 05/2020.
17 A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of Javascript. In *ECOOP*, 2010.
18 S. Holm Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *ESEC/FSE*, 2011.
19 B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Quay-De La Vallee, and S. Krishnamurthi. Modeling and Reasoning about DOM Events. In *WebApps*, 2012.
20 G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *FSE*, 2014.

21 M. Loring, M. Marron, and D. Leijen. Semantics of asynchronous javascript. In *DLS*, 2017.
22 M. Madsen, O. Lhoták, and F. Tip. A Model for Reasoning about JavaScript Promises. *PACMPL*, 1(OOPSLA):86:1–86:24, 2017.
23 M. Madsen, F. Tip, and O. Lhotak. Static Analysis of Event-Driven Node.js JavaScript Applications. In *OOPSLA*, 2015.
24 A. Almeida Matos, J. Fragoso Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *TGC*, 2014.
25 Mozilla. MDN Web Docs. `http://developer.mozilla.org/en-US/`, visited 05/2020.
26 Mozilla. Using Shadow DOM. `https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM`, visited 05/2020.
27 C. Park, S. Won, J. Jin, and S. Ryu. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *ASE*, 2015.
28 J. Gibbs Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *USENIX Security Symposium*, 2011.
29 V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015.
30 Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *ESORICS*, 2009.
31 G. Sampaio, J. Fragoso Santos, P. Maksimović, and P. Gardner. A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications (Technical Report). `https://vtss.doc.ic.ac.uk/publications/Sampaio2020Trusted.pdf`, visited 05/2020.
32 J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. JaVerT: JavaScript Verification Toolchain. *PACMPL*, 2(POPL):50:1–50:33, 2018.
33 K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE'13*, 2013.
34 G. Smith. *Local reasoning about Web programs.* PhD thesis, Imperial College, UK, 2011.
35 S. Sorhus. p-map (GitHub). `https://github.com/sindresorhus/p-map`, visited 05/2020.
36 S. Sorhus. p-map (npm). `https://www.npmjs.com/package/p-map`, visited 05/2020.
37 C. Sung, M. Kusano, N. Sinha, and C. Wang. Static DOM Event Dependency Analysis for Testing Web Applications. In *FSE*, 2016.
38 Scheme Team. The Revised Report on the Algorithmic Language Scheme. `https://schemers.org/Documents/Standards/R5RS/r5rs.pdf`, visited 05/2020.
39 The PreactJS Team. PreactJS library. `http://preactjs.com`, visited 05/2020.
40 The ReactJS Team. ReactJS library. `http://reactjs.org`, visited 05/2020.
41 The WebKit Team. WebKit Browser Engine. `https://webkit.org`, visited 05/2020.
42 Peter Thiemann. A Type Safe DOM API. In *DBPL*, 2005.
43 E. Torlak and R. Bodík. Growing Solver-Aided Languages with Rosette. In *Onward!*, 2013.
44 E. Torlak and R. Bodík. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*, 2014.
45 M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful Declassification Policies for Event-Driven Programs. In *CSF*, 2014.
46 W3C. DOM Core Level 1 Official Test Suite. `http://www.w3.org/2004/04/ecmascript/level1/core/`, visited 05/2020.
47 W3C. DOM Core Level 1 Specification. `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html`, visited 05/2020.
48 W3C. DOM Core Level 2 Specification. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`, visited 05/2020.
49 W3C. DOM Core Level 3 Specification. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`, visited 05/2020.
50 W3C. DOM Events Official Test Suite. `https://github.com/web-platform-tests/wpt/tree/master/dom/events`, visited 05/2020.
51 W3C. File API. `http://www.w3.org/TR/FileAPI/`, visited 05/2020.

52 W3C. HTML Standard. `http://html.spec.whatwg.org/multipage/workers.html#workers`, visited 05/2020.

53 WHATWG. DOM API Specification. `http://dom.spec.whatwg.org`, visited 05/2020.

54 WHATWG. The postMessage API. `https://html.spec.whatwg.org/multipage/web-messaging.html#posting-messages`, visited 05/2020.

55 K. Wheeler, S. Shaw, and F. Spampinato. `cash` (GitHub). `https://github.com/kenwheeler/cash`, visited 05/2020.

56 K. Wheeler, S. Shaw, and F. Spampinato. `cash` (npm). `https://www.npmjs.com/package/cash-dom`, visited 05/2020.

57 wpt.fyi. Events Browser Compliance. `http://wpt.fyi/results/dom/events`, visited 05/2020.

58 Y. Nashwaan. xml-js: Converter Utility between XML Text and Javascript Objects/JSON Text. `http://www.npmjs.com/package/xml-js`, visited 05/2020.

59 Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao. Virtual DOM Coverage: Drive an Effective Testing for Dynamic Web Applications. *ISSTA*, 2014.