Modeling and Reasoning about DOM Events

Benjamin S. Lerner Matthew J. Carroll Dan P. Kimmel
Hannah Quay-de la Vallee Shriram Krishnamurthi
Brown University

Abstract

Web applications are fundamentally reactive. Code in a web page runs in reaction to events, which are triggered either by external stimuli or by other events. The DOM, which specifies these behaviors, is therefore central to the behavior of web applications. We define the first formal model of event behavior in the DOM, with high fidelity to the DOM specification. Our model is concise and executable, and can therefore be used for testing and verification. We have applied it in several settings: to establish some intended meta-properties of the DOM, as an oracle for testing the behavior of browsers (where it found real errors), to demonstrate unwanted interactions between extensions and validate corrections to them, and to examine the impact of a web sandbox. The model composes easily with models of other web components, as a step toward full formal modeling of the web.

1 Introduction

Modern web applications are fluid collections of script and markup that respond and adapt to user interaction. Because their programming model differs from classic desktop applications, the *analysis* of such programs is still in its infancy. To date, most efforts have focused on individual portions in isolation: huge progress has been made in clarifying the semantics of JavaScript [10, 16, 17], in modeling the tree structure of HTML [9], and in understanding the overall behavior of the browser as a runtime environment [2, 5, 14, 15, 18]. But each of these approaches ignores the crucial element of reactivity: web programming is fundamentally *event-driven*, and employs a powerful mechanism for event propagation. Perhaps counterintuitively, the JavaScript loaded in web applications is largely *inert*, and only executes when triggered by events dispatching through the HTML structure in which it resides. To paraphrase John Wheeler's famous dictum, "HTML tells events how to propagate, and events tell HTML how to evolve."

The ability to model web applications more accurately has widespread appeal. Webapps are large codebases in languages with (currently) poor support for modularity: how can we assure ourselves that a program doesn't exhibit unintended behaviors? Many webapps include semitrusted or untrusted content such as ads: how can we ensure that a program is robust in the face of the injected content's activity? And for many web-like applications, foremost among them Firefox or Thunderbird, users avidly install extensions that deliberately and deeply modify the markup and script of the underlying program: what assurance do we have that the composite program will work correctly? Even current tools that do attempt to model both the page structure and the code [3, 4, 6] are hampered by state-space explosion, as without a precise model the potential code paths grow beyond feasibility.

Instead, we propose a simple, executable, testable model of event dispatch in web applications, in the style of λ_{JS} [10, 11, 17]. Our model is engineered to hew closely to the structure of the spec [13], to build confidence in the model's adequacy. For our purposes we abstract JavaScript and model only those APIs dealing with page structure or events; the model is easily extended to include λ_{JS} directly. Likewise we represent the page structure as a simple tree in a heap; again the model can be extended with a richer tree representation [9] for further precision.

Contributions

This paper makes the following concrete contributions:

1. A short, executable, and testable model of event dispatch (Section 4.2). Writing such a model clarifies potential sources of confusion in the spec itself, provides an oracle against which implementations can be tested, and provides a foundation for future program analyses. As a case in point, systematically testing small examples in our model revealed discrepant behavior among the major browsers.

2. Simple proofs (Section 4.1) that the model upholds properties expected of the spec, such as precisely how and when a script's side effects can affect the dispatching of current and subsequent events. Because the model closely resembles the spec, such proofs lend confidence that the spec itself enjoys the same properties; thus far such claims were merely the *intent* of the lengthy, prose spec.

It also presents two initial applications of the model:

1. We examine two Thunderbird extensions to detect a real conflict between them. The model is then used to show that the fix (as implemented by one extension author) currently suffices to correct the bug, that another, simpler fix should be more robust, and this simpler fix in turn reveals a bug in Gecko (Section 4.3).
2. We re-examine the assumptions of ADsafe [1] in light of event dispatch, to determine whether ADsafe widgets may affect the control flow of their host despite the ADsafe sandbox, and suggest directions for more robust widgets (Section 4.4).

2 Web Program Control Flow Unpacked

An intuitive but incomplete model for programming web pages is that of an asynchronous event loop. In this model, events are triggered by user interaction, and event callbacks have access to an object graph representing the tree structure of the HTML, known as the Document Object Model (DOM). The full definition of "the DOM" is, however, spread over many specifications [12, 13, 19, 21, among others], comprising far more than just this tree structure. In reality, the DOM object graph is more interconnected than a mere tree and can be arbitrarily entangled with the JavaScript heap; event callbacks are attached directly to these DOM nodes; and while the event loop itself is not available as a first-class entity through the DOM, nodes may support APIs that implicitly cause further events to be dispatched or that modify the document structure.

In short, it is naïve to think of the execution of a web program as merely an event loop alongside a tree-structured data store. Rather, the structure of the document influences the propagation of events, and the side effects of events can modify the document. Understanding web program behavior therefore requires modeling all the subtleties of event dispatch through the DOM. Like all portions of web-related programming, the event mechanisms were developed over time, resulting in historical quirks and oddities. We explain the main features of event dispatch in this section, and enunciate design goals for our model to support, then develop our model of it in the following section.

2.1 Event Dispatch in N Easy Stages

Static document structure, one event listener We take as a running example a simple document fragment of three nodes: ⟨**div**⟩⟨**p**⟩⟨**span**/⟩⟨/**p**⟩⟨/**div**⟩. In the simplest case, suppose as the page loads we attach a single event listener to the ⟨**span**/⟩:

```
spanNode.addEventListener("click",
  function(event) { alert("In click"); });
```

This statement registers the function as a *listener* for mouse "click" events only; any other event types are ignored. When an event is *dispatched* to a particular *target*, the listener on that target for that event type—if there is one—is invoked. Thus a "click" event targeted at the ⟨**span**/⟩ will yield the alert; a "keypress" event will not, nor will a "click" event targeted at the ⟨**p**/⟩ node.

Note that scripts can construct new event objects programmatically and dispatch them to target nodes. These events behave identically to browser-generated events, with one caveat addressed later.

Design Goal 1': Every node has a map of installed listeners, keyed by event type. *(To be refined)*

Multiple listeners and the propagation path We now expand the above model in two key ways. First, the suggestively named `addEventListener` API can in fact be used repeatedly, for the same node and the same event type, to add *multiple* listeners for an event. These listeners will be called in the order they were installed whenever their triggering event is dispatched. This flexibility allows for cleaner program structure: clicking on a form button, say, might trigger both the display of new form fields and the validation of existing ones; these disparate pieces of functionality can now be in separate listeners rather than one monolithic one.

Second, web programs frequently may respond to events on several elements in the same way. One approach would be to install the same function as a listener on each such element, but this is brittle if the page structure is later changed. Instead, a more robust approach would install the listener once on the nearest common ancestor of all the intended targets. To achieve this, event dispatch will call listeners on *each ancestor of the target node* as well, known as the *propagation path*. Thus adding a listener to the other two nodes in our example:

```
function listener(event) {
  alert("At " + event.currentTarget.nodeName
        + " with actual target "
        + event.target.nodeName);
}
pNode.addEventListener("click", listener);
divNode.addEventListener("click", listener);
```

and then clicking in the ⟨**span**/⟩ will trigger *three* alerts: “In click”, “At p with actual target span”, and “At div with actual target span” in that order: the event *bubbles* from the target node through its ancestors to the root of the document.[1]

For symmetry, programs may want to perform some generic response *before* the event reaches the target node, rather than only after. Accordingly, event dispatch in fact defines a so-called *capturing* phase, where listeners are called starting at the root and propagating down to the target node. To install a capture-phase listener, `addEventListener` takes a third, boolean `useCapture` parameter: when true, the listener is for capturing; when missing or false, the listener is for bubbling.

Event dispatch therefore comprises three phases: “capture”, from root to the target’s parent and running only capture-phase listeners; “target”, at the target node and running all listeners; and “bubble”, from the target’s parent to the root and running only bubble-phase listeners. The `event` parameter to each listener contains three fields indicating the current `eventPhase`, the `currentTarget`, and the intended `target` of the event. For our running example, an event targeted at the ⟨**span**/⟩ will call listeners

1. On ⟨**div**/⟩ for phase `capture`, then
2. On ⟨**p**/⟩ for phase `capture`, then
3. On ⟨**span**/⟩ for phase `target`, then
4. On ⟨**p**/⟩ for phase `bubble`, then
5. On ⟨**div**/⟩ for phase `bubble`.

Design Goal 1”: Every node has a map of installed listeners, keyed by event type and phase. *(To be refined)*

Design Goal 2: Dispatch takes as input a node and its ancestor chain, which it will traverse twice.

Aborting event propagation It may be the case that a capture- or target-phase listener completely handles an event, and that the app has no need to propagate the event further. The app could maintain some global flag and have each listener check it and abort accordingly, but this is tedious and error-prone. Instead, the `event` object can be used to stop event propagation in two ways:

- `event.stopPropagation()` tells dispatch to terminate as soon as all listeners on the current node complete, regardless of whether listeners are installed on future nodes of the propagation path. Thus calling this in a target-phase listener on ⟨**span**/⟩ will abort dispatch between steps 3 and 4 above.
- `event.stopImmediatePropagation()` tells dispatch to terminate as soon as the current listener returns, regardless of whether other listeners are installed on this or future nodes in the propagation path. Thus calling this in a capture-phase listener on ⟨**p**/⟩ will abort dispatch in the middle of step 2, even if there are more capture-phase listeners on ⟨**p**/⟩.

Design Goal 3: Dispatch can be aborted early.

Dynamic document structure: no effect! So far our example listeners have had no side effects; in general, however, they often do. This may interact oddly with the informal definitions above: for instance, if a target-phase listener removes the target node from the document, what should the propagation path be? Several options are possible; the currently specified behavior is that the propagation path is *fixed* at the beginning of dispatch, and is unmodified by changes in document structure. Thus in our running example, regardless of whether nodes are deleted, re-parented or otherwise modified, the five steps listed are unaffected.

Design Goal 4: The ancestor chain input to Design Goal 2 is immutable.

Dynamic listeners: some effect! We can now address the last oversimplification, that event listeners are added once and for all at the start of the program. In fact they can be added and removed dynamically (using the analogous `removeEventListener` API) throughout the program’s execution. For example, a common idiom is the “run-once” listener that removes itself the first time it runs:

```
function runOnce(event) {
  node.removeEventListener("click", runOnce);
  ...
}
node.addEventListener("click", runOnce);
```

Such actions have a limited effect on the current dispatch: listeners added to (resp. removed from) a *future* node in the propagation path will (resp. will not) be called by the dispatch algorithm; listeners added to (resp. removed from) the *current* or *past* nodes in the propagation path will be ignored (resp. will still be called). More intuitively, a refinement of the five steps above says that dispatching an event to ⟨**span**/⟩ will:

1’. Determine the capture-phase listeners on ⟨**div**/⟩ and run them, then

2’. Determine the capture-phase listeners on ⟨**p**/⟩ and run them, then

3’. Determine the target-phase (i.e., all) listeners on ⟨**span**/⟩ and run them, then

[1] Additionally, for legacy reasons it also propagates to the global `window` object; this detail does not substantially change any of our subsequent modeling.

4’. Determine the bubble-phase listeners on ⟨**p**/⟩ and run them, then

5’. Determine the bubble-phase listeners on ⟨**div**/⟩ and run them.

Since the determination of the relevant listeners is lazily computed in each step, dispatch will only notice added or removed listeners that apply to later steps.

Design Goal 5: The listener map is mutable during dispatch, but an immutable copy is made as each node is reached.

Dealing with legacy “handlers” Unfortunately, the mechanism explained so far—multiple listeners, capturing and bubbling, and cancellation—was not the first model proposed. Originally, authors could write ⟨**span** onclick=“`alert("In onclick");`”/⟩, and define an event *handler* for the “click” event. There can be at most one handler for a given event on a given node, which takes the form of a bare JavaScript statement.

To incorporate this legacy handler mechanism into the listener model above, handlers are implicitly wrapped in `function(event) { ... }`[2] and their return values are post-processed to accommodate the ad-hoc nature of legacy handler support. Handlers can be altered by modifying the `onclick` content attribute or by modifying the `onclick` property of the node:

```
node.setAttribute("onclick",
         "alert('New handler');");
node.onclick =
  function(event) { alert("New handler"); }
```

and for legacy compatibility, these mutations must not affect the relative execution order of the handler and any other “click” listeners.

Design Goal 1: Every node has a map of installed listeners and handlers, keyed by event type and phase.

Default actions Finally, browsers implement a great deal of functionality in response to events: clicking a link will navigate the page, typing into a text box will modify its contents, selecting one radio button will deselect the others, and so forth. Such *default actions* behave mostly like implicitly-installed listeners, with a few caveats. Default actions are *not* prevented by `stopPropagation` or `stopImmediatePropagation`; instead, listeners must call `preventDefault`. Legacy handlers can return `true` (or sometimes `false`) to achieve the same effect. Also, default actions are *not* run for programmatically constructed events; these events are considered “untrusted” and cannot be used to forge user interaction with the browser.

The default action for many events is in fact to trigger the dispatch of a new event: for example, the default action of a “keydown” event will dispatch a new “keypress” event; likewise, the default action for “mouseup” is to dispatch a “click” event and possibly a “doubleclick” event. Note that these are new dispatches; any and all changes to the document structure made by script-installed listeners will be visible in the propagation path of these new events.

Design Goal 6: Events are equipped with a default action which is the final handler of the dispatch.

2.2 Challenges

Analyzing the full control-flow of an application is difficult enough even in ideal settings when only one developer writes the complete program. Still, a whole-program analysis is possible in principle, since the entirety of the codebase is available for inspection. On the web, however, programmers frequently include code they did not author. We consider two scenarios: the intentional inclusion of third-party code such as ads, and the unforeseeable injection of user-installed extensions.

2.2.1 Invited third-party code

A typical webapp may include ads sourced from various third parties, a Twitter or blog feed, social network sharing operations, and so on. These all take the form of some user-visible UI, and nearly always include additional scripts to make the UI interactive. But such inclusion can have several unpleasant side-effects. The obvious security consequence, in the worst case when the webapp takes no precautions, is that the inserted content runs in the same JavaScript context as the webapp, with the same permissions, and can inadvertently or maliciously break the webapp. Fortunately, several frameworks exist to mitigate such naïve mistakes: tools like ADsafe [1] or Caja [20] attempt to sandbox the inserted content, isolating it within a subtree of the surrounding document and within a restricted JavaScript environment. But these also have weaknesses in the face of DOM events, as we discuss in Section 4.4.

2.2.2 Uninvited third-party code

Virtually every major browser now permits the installation of extensions. These are specifically intended for users to modify individual webapps or the browser itself. For example, there are app- or site-specific extensions that, say, supplant existing webmail auto-completion methods, or replace ads with contacts, or customize the UI of a particular newspaper or social networking site. While these extensions are sometimes written by the creators of the original application or site, in other cases they

[2] The expert reader will note that some contortions are needed to supply the right `this` object and scope to the handler.

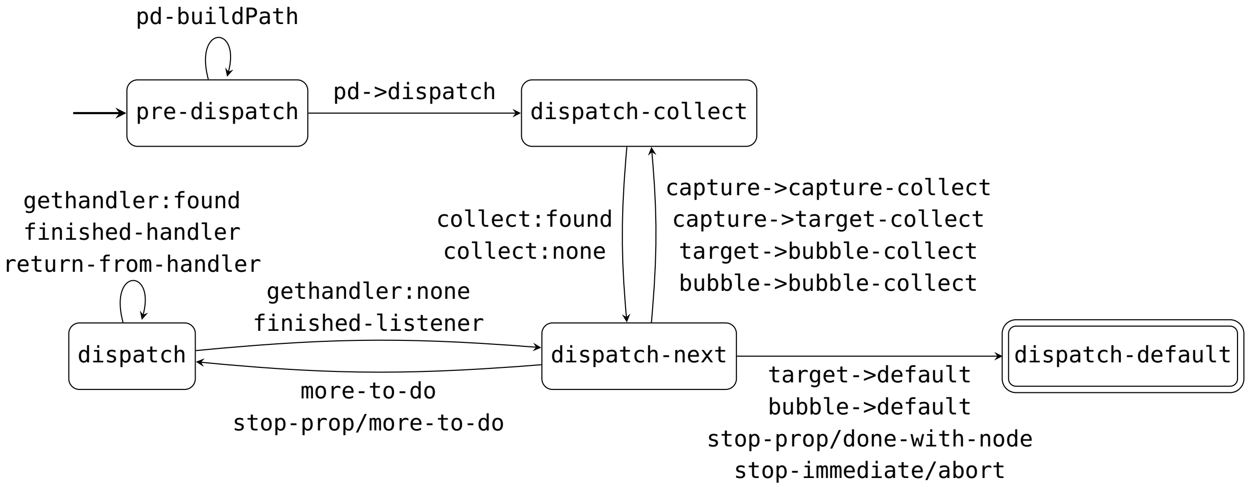


Figure 1: The core reduction steps in our model, implementing the event dispatch state machine.

are written by third parties. Other browser extensions personalize the browser's whole look-and-feel. All these extensions can be highly invasive to apps, and there is no way for app authors to anticipate these modifications. Instead, they must code defensively in *all* event listeners, for which they need a model of what to defend against.

3 Modeling DOM Events

Having informally laid out how event dispatching works, we are ready to model it precisely. We will first describe the model itself, then explain how we account for its relationship to the actual DOM specification. Design goals 1, 3, and 6 are used to construct the model; the other three express properties about that model that we prove in Section 4.1. Section 5 presents extensions to the model.

3.1 Model Highlights

Because the DOM is essentially a large automaton that determines what operations will execute next, we model it using an operational semantics. In particular, because of the ability to abort dispatch in subtle ways (see Design Goal 3), we find it most effective to use the evaluation context style of Felleisen and Hieb [8], which was initially designed to model control operators (such as exceptions and continuations) in regular programming languages and is thus well suited for that purpose.

Our full model, which can be found at `http://www.cs.brown.edu/research/plt/dl/domsemantics/`, is 1200 lines of commented code. It is implemented using the PLT Redex modeling language [7], which provides programming environment support for models in the Felleisen-Hieb style. Here we present the highlights that will help the reader navigate that document.

3.1.1 Stages of a Dispatch

The Events spec defines the procedure for synchronously dispatching a single event in careful detail, and the prose is full of challenging nuances. Conceptually, however, the spec defines a single event dispatch as an automaton with five states. The states and their transitions, as named in our model, are shown in Fig. 1; we discuss the key transitions below. Our model identifies eight transitions, with eighteen triggering conditions: a reasonable size, given the many interacting features of event dispatch, and certainly more concise than the original spec.

1. Determining the propagation path. Event dispatch begins by determining the propagation path for the event: the ancestors of the target node at the time dispatch is initiated. Our model builds this path in the `pre-dispatch` state. The spec states that "once determined, the propagation path must not be changed," regardless of any page mutations caused by listeners that are triggered during dispatch *(Design Goal 1)*. This is trivially maintained by our model: every transition between the `dispatch-next`, `dispatch-collect` and `dispatch` states (described below) preserve the path without modification.

2. Determining the next listener. The flow of an event dispatch may be truncated in one of three ways: after the completion of the current listener, after the completion of any remaining listeners on the current node and phase, or the default action may be canceled. Further, some events may skip the bubble phase entirely. When any given listener completes execution, the dispatch algorithm must check whether any of these truncations have been signaled, and abort dispatch accordingly *(Design Goal 3)*. If none have, then dispatch proceeds to the next listener

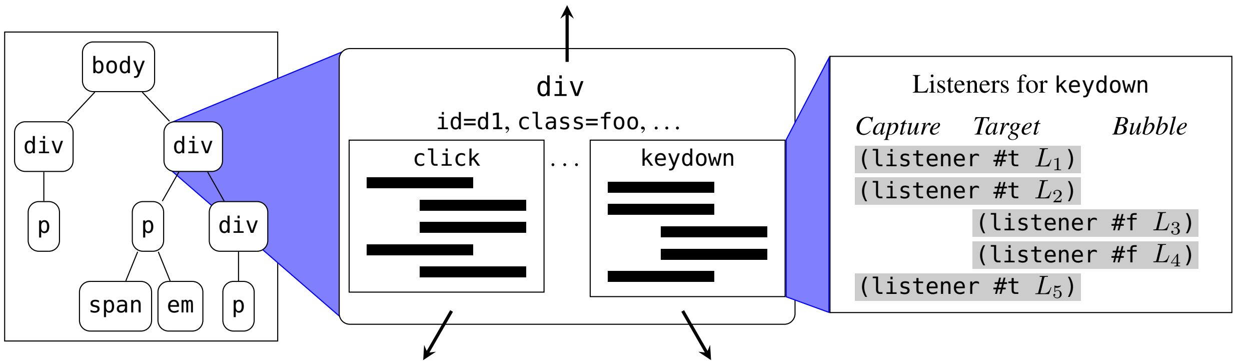


Figure 2: Schematic representation of the model's store. The store contains nodes connected in a tree structure (left). Each node stores its name, attributes, pointers to its parent and children, and suites of event listeners, grouped by event type (middle). Each suite contains three lists, one for each phase of dispatch; listeners for either $\langle capture \rangle$ or $\langle bubble \rangle$ phases also apply—in order—to the $\langle target \rangle$ phase (right). Each listener holds a pointer (L_i) to its actual code.

for the current node and phase or, if no such listener exists, begins collecting listeners for the next node (and possibly phase) on the propagation path. Precisely identifying these conditions is the crux of our model, which reifies them as ten transitions out of the `dispatch-next` state.

3. Determining listeners for the current node and phase. Perhaps one of the subtlest requirements of the spec determines which listeners must be called when dispatch reaches a node on a propagation path—and not all browsers currently get this right (see Section 4.2). As noted in Section 2.1, the list of listeners for a given node and phase is fixed only when dispatch reaches that node; this step is accomplished by the `dispatch-collect` model state *(Design Goal 5)*. Unfortunately here the spec conflates specification and representation: it implicitly assumes a flat list of the installed event listeners, and must include qualifiers to predicate which listeners should run. Our model avoids making assumptions about representation, using a structure that exposes the semantic intent (see Section 3.1.2 below), and merely copies the relevant list of installed listeners for the current phase into the dispatch context, thereby insuring that any changes to the installed listeners on the current node and phase *will not take effect* until a subsequent dispatch. (Still, any modifications to listeners installed on nodes for phases later in the current dispatch *will* be visible.) Accordingly, the model's transitions here are far clearer than the spec they implement.

4. Executing a listener. Either transition from `dispatch-next` to the `dispatch` state determines that a given listener should be executed. The model then records both the current listener and the remaining target nodes, and begins executing the listener body. While in this state, listeners may invoke additional, reentrant ("synchronous") event dispatches, may cancel the current event dispatch, or generally may modify the DOM however they choose. Once a `dispatch` context completes its listener body, it transitions back to `dispatch-next` to determine the next listener to call.

5. Default actions. When `dispatch-next` reaches the end of the propagation path, or when the bubble phase would begin but the current event does not bubble, the algorithm must execute the *default actions*, if any, for the given event and event target. We model this with a `dispatch-default` state *(Design Goal 6)* and a meta-function (not shown) to compute the relevant default actions. This meta-function is the only portion of the dispatch algorithm that inspects the detailed form of the event and target; everything else is agnostic. A model of the full HTML spec would supply this meta-function, specializing the event dispatch mechanism to the events applicable to HTML documents.

3.1.2 Representing Event Listener Lists

The precise storage for event listeners encodes several requirements culled from disparate portions of the spec, and embodies Design Goal 1. We give the precise type in Fig. 3b, and explain its design in four stages.

First, the spec mandates that event listeners installed for the same node, event type and dispatch phase must be called in the order they were installed. Accordingly, every node contains a map (LS) from event type and phase to a vector of listeners.

Second, the spec elsewhere states listeners may be installed for either $\langle capture \rangle$ and $\langle target \rangle$ phases, or $\langle target \rangle$ and $\langle bubble \rangle$ phases. At first glance, it seems that we might simply maintain separate lists for $\langle capture \rangle$- and $\langle bubble \rangle$-phase listeners, but that runs afoul of the ordering requirement when dispatch reaches the $\langle target \rangle$ phase. Instead, we must also maintain a list of $\langle target \rangle$-phase listeners, and when adding a new listener, we must update two lists in our map: this is accomplished by the `addListener` meta-function.

Third, the spec requires that the triple of arguments $(\langle eventType \rangle, \langle usesCapture \rangle, \langle listener \rangle)$ must be unique within each node: while a given function may be installed both as a capture-phase listener and as a bubble-phase listener on the same node, subsequent installations will have no effect. In combination with the previous requirement, one implicit consequence is that a function may be called *twice* during the `target` phase; though true, this is not immediately obvious from the spec wording, but is evident from our rules.

Finally, the spec defines how event listeners may be removed from a node: again, from both capture and target phases, or from both target and bubble phases. Thanks to the uniqueness requirement and our modeling of `addEventListener`, we know that a given listener may be present twice in the target-phase list, so we must record *which* target-phase listeners were also installed on the capture phase, and which were not, or else we might remove the wrong listener and violate the ordering requirement. Consider the following program fragment:

```
node.addEventListener("click", true, f1);
node.addEventListener("click", true, f2);
node.addEventListener("click", false, f1);
node.removeEventListener("click", true, f1);
```

While it is intuitively clear that the call to `removeEventListener` must remove `f1` in the `capture` phase, it must also remove the *corresponding* `f1` in the target phase, i.e., the first one.

Remark: In prior work [15], in which the first author implemented the event dispatch algorithm, he read the documentation for `addEventListener` too quickly; it is excerpted in Fig. 3a. Note the emphasized text: in fact, the specification is inconsistent in defining on which phases listeners may be installed! By contrast, the meta-function in Fig. 3b uses the `useCapture` flag exactly once, and hence avoids and resolves this error.

The store as described here is redundant: the $\langle target \rangle$ list by itself contains sufficient information to produce the spec-defined behavior. However, this redundancy is intentional: it simplifies the determination of relevant listeners (the `dispatch-collect` state earlier), emphasizes the "doubled" effect of `addEventListener`, and indirectly encourages implementers to treat the model as a specification rather than an implementation guide.

3.2 Modeling Challenge: Adequacy

Whenever researchers build a model of a system, they must demonstrate why the model adequately represents the system being examined, or else the model is of no relevance. This is an inherently informal process, as the system here is the prose of the specification; if the spec were amenable to formal methods in the first place, there would be no need for a formal model!

To simplify the case for our model's adequacy, we have annotated each paragraph of the spec with a link to the relevant definitions and reduction rules of our model. A reasonably knowledgeable reader could flip back and forth between the spec and the model, and convince herself that the model faithfully represents the intent of the spec. An excerpt of this is shown in Fig. 3, where we show the spec's definition for `addEventListener` and the corresponding Redex metafunction that installs the listener into our model.[3]

Of course, the DOM also lives through many implementations. We can therefore test our model to determine whether it conforms to the behavior of actual implementations. We have begun doing so, and discuss the results in Section 4.2. Ultimately, we have to choose between modeling a specific browser or the spec; we have chosen the latter, but the decisions we have made are localized and can thus be altered to reflect one particular implementation, if desired.

4 Applications

We now demonstrate the utility of our model by discussing its application in various settings.

4.1 Provable Properties of the Model

We concern ourselves here only with *well-formed* states of the model: a finite statement/heap pair (S, H) is well-formed if

1. There are no dangling pointers from S into H.
2. The heap is well-typed: Every heap location mentioned in S is used consistently either as a node or as a listener.
3. There are no dangling pointers within H: the parents and children of every node n must be present in H.
4. The nodes in the heap are tree-structured: No node is its own ancestor, descendant or the child of two distinct nodes.
5. For every listener (`listener` b L) or handler (`handler` L), L points to a statement S' and (S', H) is well-formed.

[3] The spec requirement that `addEventListener` be idempotent is in fact defined elsewhere; that text in turn corresponds to the (elided) `addListenerHelper` metafunction.

addEventListener

> Registers an event listener, depending on the `useCapture` parameter, *on the capture phase of the DOM event flow or its target and bubbling phases.*
>
> **Parameters:**
>
> > **type** of type `DOMString`: Specifies the `Event.type` associated with the event for which the user is registering.
> >
> > **listener** of type `EventListener`: …
> >
> > **useCapture** of type `boolean`: If true, `useCapture` indicates that the user wishes to add the event listener *for the capture and target phases only*, i.e., this event listener will not be triggered during the bubbling phase. If false, the event listener must *only be triggered during the target and bubbling phases.*

(a) Excerpt from the specification of `addEventListener`; emphasis added to highlight self-inconsistencies.

$$
\begin{array}{rrcl}
P \in & \textsc{Phase} & ::= & \langle \mathit{capture} \rangle \mid \langle \mathit{target} \rangle \mid \langle \mathit{bubble} \rangle \\
L \in & \textsc{Listener} & ::= & \texttt{listener}\ S \\
S \in & \textsc{Stmt} & ::= & \texttt{skip} \mid \texttt{return}\ \mathit{bool} \mid S; S \\
 & & \mid & \texttt{stop-prop} \mid \texttt{stop-immediate} \\
 & & \mid & \texttt{prevent-default} \\
 & & \mid & \texttt{addEventListener}\ N\ T\ \mathit{bool}\ L \\
 & & \mid & \texttt{remEventListener}\ N\ T\ \mathit{bool}\ L \\
 & & \mid & \texttt{debug-print}\mathit{string} \\
T \in & \textsc{EvtType} & ::= & \texttt{"click"} \mid \texttt{"keydown"} \mid \cdots \\
\mathit{LS} \in & \textsc{LMap} & ::= & (T \times P) \rightharpoonup (\overrightarrow{\mathit{bool} \times L}) \\
N \in & \textsc{Node} & ::= & (\texttt{node}\ \mathit{name}\ \mathit{LS}\ \ldots)
\end{array}
$$

```
(define-metafunction DOM
  [(addListener
      LS string_type bool_useCapture L)
   (addListenerHelper
     (addListenerHelper
       LS string_type target L)
     string_type
     ,(if (term bool_useCapture)
        (term capture)
        (term bubble))
     L)])
```

(b) Excerpt from our Redex model of `addEventListener`. Note that the impact of the `useCapture` is defined exactly once, leaving no room for self-inconsistency.

Figure 3: Defining and modeling `addEventListener`

We can now prove that our model upholds the invariants stated in our Design Goals:

Theorem 1. *Once computed, the event propagation path is fixed for each dispatch.* (Design Goal 4)

Proof sketch. By inspection of the reduction rules: every rule between `dispatch-next`, `dispatch-collect` and `dispatch` leaves the path unchanged. Rules leading to `dispatch-default` vacuously leave the path unchanged, since dispatch has passed the end of the path. The remaining rules in `pre-dispatch` compute the path itself. □

Theorem 2. *During dispatch, once the event listener list for a given node and phase is computed, it is unaffected by calls to `addEventListener` or `removeEventListener` in any invoked listeners.* (Design Goal 5)

Proof sketch. This is the express purpose of `dispatch-collect`: only it examines the store to collect the currently installed listeners, and copies that list into the `dispatch-next` context. All further reductions from `dispatch-next` use the copy, and are unaware of any changes in the store. □

Theorem 3. *Event dispatch is deterministic.*

Proof sketch. By inspection of the reduction rules: the left hand sides of the rules never overlap. □

Additionally, we can prove several other key properties:

Lemma 1. *Preservation of well-formedness: given a well-formed (S, H) such that $(S, H) \rightarrow (S', H')$, (S', H') is well-formed.*

Lemma 2. *Progress: a well-formed term is either $(\texttt{skip}, H)$ or it can take a step via $(\rightarrow)$.*

Theorem 4. *Termination: assuming all listeners and handlers terminate, and do not recursively dispatch events, every well-formed event dispatch completes: $((\texttt{pre-dispatch}\ loc_n\ \texttt{()}\ e), H) \rightarrow^* (\texttt{skip}, H')$.*

Proof sketch. The propagation path of any node in a well-formed state is finite, because the heap is finite and tree-structured. Every transition from `dispatch-next` either reduces the number of remaining listeners on the current node, or the number of remaining nodes in the propagation path. Every transition out of `dispatch-collect` returns to `dispatch-next` in a single step. By assumption, every listener or handler terminates (in our minimal language this is trivial; in full JavaScript it is not), so every transition to `dispatch` will return to `dispatch-next` in finite time. Finally, again by assumption, the default handlers run by `dispatch-default` terminate. □

Theorem 5. *Double dispatch* (Design Goal 2)*: for every node on a well-formed propagation path that is not the target node, if dispatch is not stopped then that node will be visited exactly twice, once for capture and once for bubbling phases.*

Proof sketch. Because the heap is tree-structured, and by construction, every node in the propagation path is distinct. By construction, `pd->dispatch` collects listeners for the root node; all subsequent transitions from `dispatch-next` to `dispatch-collect` collect listeners for the remaining nodes on the path except the target (`capture->capture`), then the target (`capture->target`), then the path is traversed backward (`target->bubble` and `bubble->bubble`). By immutability of the path, every node visited in capture phase is therefore visited again in bubble phase. □

These properties are intuitively expected by the authors of the dispatch spec, and formally hold of our model; the adequacy of our model (Section 3.2) implies that these properties do hold of the spec itself.

4.2 Finding Real-World Inconsistencies

Our proof of determinism, combined with the model's adequacy, is tantamount to stating that the spec is unambiguous. We have not encountered any obvious ambiguities; our reading of the spec assigned to every claim a specific interpretation (see Section 3.2). But we nevertheless observe differing behavior on real browsers.

Erroneous Treatment of `removeEventListener` We can use our model to randomly construct test cases, or systematically generate a suite of related test cases, observe their behavior in the model, then translate the tests to HTML and JavaScript and test existing browsers to see if they match our expectations. We have been doing so, and continue to test ever larger instances of the model against browsers.

During testing, we have already found several divergences between our model and real browsers, which further revealed inconsistencies between major browsers themselves. The simplest example appears in a systematic suite checking the handling of `removeEventListener`. These tests all use the same $\langle\textbf{div}\rangle\langle\textbf{p}\rangle\langle\textbf{span}/\rangle\langle/\textbf{p}\rangle\langle/\textbf{div}\rangle$ document, and install the following listeners:

```
var targetNode, targetCapture;
var triggerNode, triggerCapture;
function g(event) { alert("In g"); }
function f(event) {
  targetNode.removeEventListener("click",
     g, targetCapture);
}
triggerNode.addEventListener("click", f,
                              triggerCapture);
targetNode.addEventListener("click", g,
                             targetCapture);
```

In words, this installs listener `f` on `triggerNode`, either for capture (`triggerCapture` $=$ *true*) or not, that will then remove `g`. It then installs listener `g` on `targetNode` for capture or not (`targetCapture`). A systematic search of all possible values of these four variables reveals that when `targetNode` $=$ `triggerNode` $\neq \langle\textbf{span}/\rangle$ and `targetCapture` $=$ `triggerCapture`, browser behavior differs. Chrome (v15 and v16), Safari (v5.0.1) and Firefox (v3.6 through v8) will *not* execute `g`, while Internet Explorer (v9) and Opera (v11) *will*.

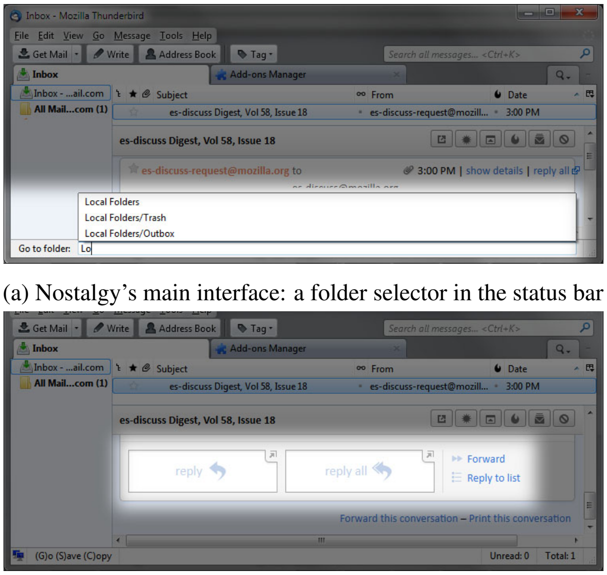
Our model predicts the latter behavior is correct, but in truth one of three situations may hold: IE, Opera and our model may be right, or Chrome, Safari and Firefox may be right, or all six may be wrong. Regardless, our model simplifies making testable predictions about browsers.

Treatment of Legacy Handlers Another example probes the corner cases surrounding setting and clearing legacy handlers. Within a few minutes of generating tests, we found examples where our model disagrees with IE, Chrome and Firefox—and the browsers all disagree with each other, too. Here, the events spec delegates responsibility to the HTML 5 spec itself, which defines how the setting and clearing of handlers interacts with existing listeners. Browsers, however, appear to have implemented variations on the specified behavior. We have identified two variations each for setting and clearing handlers; our model can accommodate them by changing the setting or clearing rule, without needing changes anywhere else. Further testing is needed to decide which of these variations, if any, corresponds to each browser's behavior. More broadly, by continuing to run such tests, we hope to build greater confidence in the quality of the model (and, perhaps, improve the uniformity of browsers, too).

4.3 Detecting Real Extension Conflicts

One of the authors routinely uses extensions to customize Thunderbird. One such extension is Nostalgy[4], which provides several convenient hot keys for archiving messages and navigating among folders. For example, pressing 'S'

[4] `http://http://code.google.com/p/nostalgy/`



(a) Nostalgy's main interface: a folder selector in the status bar

(b) Thunderbird Conversation's main interface: text boxes in the conversation view for quick replies

Figure 4: Screenshots of the conflicting extensions' UIs.

will save the current message. This is achieved by two event listeners on the Thunderbird global `window` object:

```
function onNostalgyKeyPressCapture(event) {
  // handle ESC key and cancellation
}
function onNostalgyKeyPress(event) {
  // show folder selector and handle commands
}
window.addEventListener("keypress",
          onNostalgyKeyPress, false);
window.addEventListener("keypress",
          onNostalgyKeyPressCapture, true);
```

Implicit in this code is the assumption that all key presses are intended to control Thunderbird, and not, say, to input text. However, another extension, Thunderbird Conversations[5], redefines the email preview pane to show a Gmail-like conversation view, complete with "quick reply" boxes where the user can compose a response without leaving the main window. This functionality is implemented by the default actions of the quick-reply ⟨**textarea**/⟩ tags, along with a bubble-phase listener on their grandparent ⟨**div**/⟩:

```
quickReplyDiv.addEventListener("keydown",
  function convKeyDown(event) {
    // ENTER=>send message, ESC=>cancel
    event.stopPropagation();
  });
```

At first glance, nothing in this code appears problematic; indeed, Conversations and Nostalgy are listening to two different events. However, our model includes the fact that the default action of a "keydown" event is to dispatch a "keypress" event, and while Conversations does `stopPropagation`, it does not `preventDefault`—which means that any key strokes typed into the quick-reply box will effectively call `convKeyDown`, `onNostalgyKeyPressCapture` and finally `onNostalgyKeyPress`. Consequently, typing a word containing an 'S' will steal focus from editing the message, and jump to Nostalgy's "Save Message" UI! And indeed, if we input the Thunderbird DOM and these three event listeners into our model, it confirms that this behavior is correct according to the event dispatch rules.

The author reported this bug to Conversations' developer, who produced the following fix:

```
quickReplyDiv.addEventListener("keypress",
  function convKeyPress(event)
    { event.stopPropagation(); });
quickReplyDiv.addEventListener("keyup",
  function convKeyUp(event)
    { event.stopPropagation(); });
```

Adding these two listeners to our model shows that event dispatch now calls `convKeyDown`, `onNostalgyKeyPressCapture`, and `convKeyPress`, and no longer calls `onNostalgyKeyPress`, thereby avoiding the bug for now. However, some of Nostalgy's code still gets called, leaving open the potential for future bugs. Examining the model, and recalling that Nostalgy never expected to "see" keypress events due to text input, notice that Nostalgy's code is called only because the "keypress" is dispatched, which occurs only because of the "keydown" default action. A simpler, and more robust, fix is therefore available to Conversations: adding a call to `preventDefault` in `convKeyDown` would prevent the dispatch of the "keypress" event in the first place. Implementing this approach in our model confirms that the "keypress" event is never fired. However, implementing it in Conversations does not work, and instead reveals a bug in Thunderbird: "keypress" events appear to be dispatched *regardless* of whether the default has been prevented or not, contrary to the spec.

Generalizing from this example, we can annotate listeners in our model with provenance information, and then query the model for whether there exist any $(eventType, targetNode)$ pairs for which dispatch will cause control to flow from one extension's listeners to another's. We anticipate that such queries will statically yield other pairs of extensions whose behavior might conflict; for example, Conversations is known to be incompatible with other hotkey-related extensions; this analysis can reveal others, then pinpoint where bugfixes are needed.

[5] https://github.com/protz/GMail-Conversation-View/

4.4 Event Propagation and Sandboxes

We have previously discussed the use of sandboxes to protect webapps against invited third-party code. Such sandboxed content, or “widgets” as they are often called, should not be able to suborn the surrounding document, and in fact, some successful efforts have *proven* the effectiveness of these sandboxing techniques [17]. However, there is a serious weakness in these proofs. The authors caveat their results as applying only over the DOM portion they model—which does not include events. While the sandboxes may successfully prevent widgets from calling DOM methods or executing arbitrary code, event dispatch provides an indirect way for widgets to invoke portions of the code of the webapp.

Specifically, because widgets present some form of UI, they can be the target of user-generated events. As a hypothetical example, a malicious widget might display what looks like a typing game. Because of the event dispatch rules, those keystrokes can bubble out of the widget and potentially invoke listeners higher in the webapp’s document. The widget could selectively call `stopPropagation`, filtering out unwanted letters, and thereby forge an input to the webapp that the user did not intend. Worse, even if the sandbox stopped *all* propagation, it cannot prevent against listeners being invoked during the capture phase, which means the widget is getting to execute code as the program. To date, we know of no modeling effort that attempts to prove that widgets are sandboxed from conducting such a spoofing attack.

There are two possible approaches to protect against this. First, a conscientious webapp developer can protect his application against such unwanted events with defensive code that checks—in every event handler in the propagation path of a widget—whether the target of the event is in their own content or in the widget. Such coding practices are onerous and fail-open: one missed check could suffice for the attack to proceed. And yet, hardening every event listener would preclude extensions from integrating properly into the webapp, as any events originating from their UI would summarily be ignored! Second, the sandbox could decide that, because it cannot truly protect against a malicious widget due to the capture phase, it might choose to implement its own event dispatch model (as some libraries like jQuery[6] do). In such cases, the sandbox or library developer undertakes the burden of establishing properties of their custom event model. In either case, our DOM model would be useful: in the former case, to determine what propagation cases the surrounding page’s listeners are missing, and in the latter case, by being a basis for formalizing and then proving properties about the custom event model.

5 Related and Future Work

We have already introduced most of the related work in earlier sections of the paper. In particular, Featherweight Firefox [5] and Featherweight DOM [9] present the first formal models of a browser and of DOM tree update, while Akhawe et al. [2] model some security-relevant events but not their dispatch. This work sits between the three and reasons about the reactive behavior of web pages. There are several avenues for future improvement:

Keeping pace with moving standards While building our model, we were made aware of the DOM4 draft spec [21], which will supercede the Level 3 Events spec we modeled. At this time, the draft is not complete enough to model, though it is not intended to introduce substantive changes. Our model should carry over nearly unchanged.

Incorporating JavaScript Fully To focus on the DOM, we have represented JavaScript procedures with a simplified statement language for manipulating listeners and handlers (Fig. 2). This is sufficient for many practical modeling purposes, but it fails to fully capture the effect of JavaScript, which may be needed for some analyses.

Fortunately, this is easy to remedy. Our Redex model is formulated such that it will be straightforward engineering to incorporate the Redex model of λ_{JS}. In particular, though presented as states here, the five steps of event dispatch are modeled as contexts, which provides a great deal of flexibility. λ_{JS} models all of JavaScript with evaluation contexts [8], including one for function calls. In essence, event dispatch is a baroque form of a calling context, namely one that invokes multiple functions in sequence based indirectly on the DOM and the current event, rather than a simple function pointer. Our DOM model will change slightly to incorporate a reified event object, rather than just data carried in the `dispatch-*` contexts; we foresee no technical hurdles here.

We can therefore enhance our model by discarding the simplified statement language in favor of true JavaScript statements. Doing so brings significant benefit to JavaScript analyses as well. Without the structure provided by our model, an analysis of a JavaScript program would necessarily miss many flows that are not caused by explicit function calls in the program text.

Modeling the Document Tree Naturally, the precision of analyses is limited by the precision of modeling the document’s structure. We currently model the tree merely as a set of nodes connected by pointers: nothing explicitly records that the structure is a tree rather than an arbitrary graph. We have engineered our model such that it should be possible, though likely not simple, to integrate more powerful tree logics such as separation or context logic [9], and thus improve the model’s overall precision.

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

Iframes and nested documents Our model currently assumes that there is only one document under consideration. Consequently, event propagation (naturally) stops at the document root. A richer model would incorporate a notion of documents and their nesting within ⟨**iframe**/⟩ elements, and explicitly include a rule that terminates the propagation path at the document root.

Additionally, because our model does not fully model JavaScript, we do not model the `window` object, or include it as the first and last targets when constructing propagation paths. This detail does not materially affect our description of event dispatch, and is easy to include.

Non-tree-based dispatch We have focused in our model on how event dispatch proceeds with tree-based sources of events, as they dominate other event sources. However, newer additions to the DOM also supply events that are not dispatched along the tree. For example, `XMLHttpRequest` responses, ⟨**audio**/⟩ and ⟨**video**/⟩ status updates, and web workers all generate events as their states change. To incorporate those into our model, we need only create rules for each of them that construct their specific propagation paths, bypassing `pre-dispatch` and jumping directly to the `dispatch-collect` context. From then on, dispatch proceeds as normal.

Acknowledgements

This work is partially supported by the US National Science Foundation and Google. We are grateful for feedback about this work from Joe Politz and Arjun Guha. We thank Robby Findler for his help getting our model off the ground, and his co-authors and him for their excellent PLT Redex semantics modeling tool.

References

[1] ADsafe. Retrieved Nov. 2009. `http://www.adsafe.org/`.

[2] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium* (2010).

[3] Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M. VEX: vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium* (Berkeley, CA, USA, Aug. 2010), USENIX Association, pp. 22–22.

[4] Barth, A., Felt, A. P., Saxena, P., and Boodman, A. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)* (2010).

[5] Bohannon, A., and Pierce, B. C. Featherweight Firefox: formalizing the core of a web browser. In *USENIX Conference on Web Application Development (WebApps)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 11–11.

[6] Djeric, V., and Goel, A. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 23–23.

[7] Felleisen, M., Findler, R. B., and Flatt, M. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[8] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103*, 2 (1992), 235–271.

[9] Gardner, P. A., Smith, G. D., Wheelhouse, M. J., and Zarfaty, U. D. Local Hoare reasoning about DOM. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (New York, NY, USA, 2008), ACM Press, pp. 261–270.

[10] Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)* (Berlin, Heidelberg, 2010), Springer-Verlag, pp. 126–150.

[11] Guha, A., Saftoiu, C., and Krishnamurthi, S. Typing local control and state using flow analysis. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software* (Berlin, Heidelberg, 2011), ESOP'11/ETAPS'11, Springer-Verlag, pp. 256–275.

[12] Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S. Document object model (DOM) level 3 core specification. Written Apr. 2004. `http://www.w3.org/TR/DOM-Level-3-Core/`.

[13] Ian Hickson, E. HTML5: A vocabulary and associated APIs for HTML and XHTML. Retrieved July 6, 2011. `http://dev.w3.org/html5/spec/Overview.html`.

[14] Lerner, B. S. *Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design*. PhD thesis, University of Washington Computer Science & Engineering, Aug. 2011.

[15] Lerner, B. S., Burg, B., Venter, H., and Schulte, W. C3: An experimental, extensible, reconfigurable platform for HTML-based applications. In *USENIX Conference on Web Application Development (WebApps)* (Berkeley, CA, USA, June 2011), USENIX Association.

[16] Maffeis, S., Mitchell, J. C., and Taly, A. An operational semantics for javascript. In *Asian Symposium on Programming Languages and Systems (APLAS)* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 307–325.

[17] Politz, J. G., Eliopoulos, S. A., Guha, A., and Krishnamurthi, S. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 12–12.

[18] Reis, C. *Web Browsers as Operating Systems: Supporting Robust and Secure Web Programs*. PhD thesis, University of Washington, 2009.

[19] Schepers, D., and Rossi, J. Document object model (DOM) level 3 events specification. Written Sept. 2011. `http://dev.w3.org/2006/webapi/DOM-Level-3-Events/html/DOM3-Events.html`.

[20] The Caja Team. Caja. Written Nov. 2009. `http://code.google.com/p/google-caja/`.

[21] van Kesteren, A., Gregor, A., and Ms2ger. Dom4. Written Jan. 2012. `http://dvcs.w3.org/hg/domcore/raw-file/tip/Overview.html`.