Language Support for Extensible Web Browsers

Benjamin S. Lerner
University of Washington
blerner@cs.washington.edu

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

Web browsers are sophisticated and crucial programs, and millions of users *extend* their browsers to customize their browsing experience. In this paper we argue the position that such extensions themselves constitute an important facet of web applications—one in need of serious programming-language research attention. We illustrate this position by contrasting the extension mechanisms of the two predominant extensible browsers, Mozilla Firefox and Google Chrome, and highlighting their weaknesses. We then describe very preliminary work that addresses these shortcomings.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Extensions, Browsers, Compatibility

1. Introduction

Intuitively, an *extensible system* is one that permits later revision of the previously-designed base system: additions to, improvements upon, or replacements of existing functionality. In the past few years, people have come to expect their *web browser* to be an extensible system, adding toolbars, social-network customizations, interface tweaks, and many other personalizations to adapt the browser to their needs. Browser extensions are wildly popular: Mozilla hosts over 6,000 Firefox extensions downloaded over 1.5 billion times,[1] while Google already hosts over 3,600 extensions for Chrome.[2]

Writing individual browser extensions is not difficult, much like basing an application off an existing, rich library. These extensions frequently need to interact with the browser in fine-grained, non-trivial ways. The challenge in writing browser extensions well lies in making them robust in the face of other extensions: unlike standalone client applications, extensions do not get to monopolize the browser that hosts them. This is both a hardship and a bonus, as extensions can use this common host to cooperatively extend one another. However, currently there is no adequate support for ensuring that extensions are *compatible* with each other.

Our motivation in this paper is to promote extension development as a programming model worthy of study, and to focus attention on language support for improving extension development, and especially extension compatibility efforts. We argue that:

1. *A powerful extension mechanism for browsers is justified and desirable.* The browser itself has become an extensible system whose extensions often interact with, or are themselves, web applications. Often, these extensions need to interact with each other as well. Extensions are a hybrid: they are (pieces of) separate programs and so reasonably are self-contained entities distinct from the browser, but are also shared tenants of a browser environment and so must coexist with each other.
2. *The two browsers that currently implement extensions do so poorly.* Firefox provides a flexible and powerful framework that yields essentially no reasonable semantics or security guarantees about multiple extensions: they are as privileged and unrestricted as the browser itself. Chrome has the reverse problem: in sandboxing extensions into a reasonable security framework, it has prevented useful and fine-grained interactions among them.
3. *Programming languages research is an appropriate tool to help.* Defining how extensions may or may not interact with one another is a problem of semantics. Detecting and resolving conflicts among those interactions can benefit from declarative, language-based security techniques.

The rest of this position paper is organized as follows. Section 2 justifies the need for extensions by describing the breadth of extensions today and at a high level what browser support they require. Section 3 takes a closer look at the extension models of Firefox and Chrome, focusing on the limitations mentioned above. Section 4 presents our

[1] `http://blog.mozilla.com/addons/2009/08/11/how-many-firefox-users-use-add-ons/`

[2] `https://chrome.google.com/extensions`

APLWACA '10 June 6, Toronto, Canada.
Copyright © 2010 ACM 978-1-60558-913-8/10/06. . . $10.00

preliminary and ongoing efforts toward defining a reasonable semantics for extensions. Section 5 concludes.

2. The case for extensions

Web applications are becoming large collaborations of software spanning server-side logic and data storage, client-side code and UI elements, network interactions (social and otherwise), and more—the boundaries of such a web application are often indistinct, and in general do not fit neatly into the HTML page-as-document model of web interaction. Browser extensions form a new facet of this space that runs neither on servers in the cloud nor as web content within a browser. Rather, they run as *part* of the browser, transcending any one web page to enhance the browsing experience itself, blurring further the distinctions between client- and server-side code.

Social networks, for example, are far more about the content being generated and shared by their members than they are about the pages used to view that content. StumbleUpon[3] is a browser toolbar that lets its users group sites by their personal interests, then use those groupings to quickly find other pages matching their current interest. This experience is inherently *about* pages, rather than *within* pages: a function of the browser and not its content. Likewise, Twitter clients such as TwitterBar or Yoono focus on letting users quickly publish tweets, often in response to what they are browsing; note that these clients are written to target the Twitter protocol, but are not applications written by Twitter itself. RSS aggregators such as Feedly explicitly create client-side views of data published from multiple sources, without needing a dedicated web site such as Google Reader to view the aggregated content.

Not all browser extensions are necessarily a part of web applications as these examples are. Many "only" modify the browser's UI, or stay purely on the client's machine. Others are primarily client-based but extend into network services. Mozilla Weave, for example, synchronizes multiple clients' history, bookmarks, and passwords with cloud-backed storage: it improves the client-side experience by transparently using the network. There is no clear line separating "client" extensions from "web application" ones. Moreover, in many cases, there may not even be a clear line separating one extension from another. For example, Firebug is an extremely popular extension permitting developers to debug web content, capturing page content, script, network requests, cache behavior and more. FireDiff is another extension that explicitly extends Firebug with a new view showing diff-like traces of page events.

These seven examples demonstrate how broadly extensions may behave. Moreover, no one user will ever install all available extensions: a web developer might install Firebug and scoff at using Yoono, while a social-networker might take the exact opposite stance. Not only do extensions let users customize the browser to their own needs without bloating it for others, they let users add features that *may not have existed* when the browser was first developed.

2.1 Designing for power, flexibility and stability

Current browser designs trade extension capability and flexibility for security and stability. On the one hand, extensions should be able to customize the appearance and behavior of the browser in a fine-grained, pervasive way, while multiple extensions should be able to interact with, build upon, or complement each other. This is the essence of Firefox's approach. On the other, extensions must be restrained sufficiently that they cannot destabilize the base system; in particular, malicious extensions should not be able to subvert the browser. This is the essence of Chrome's approach.

A system such as Firefox's poses an additional stability challenge: multiple interacting extensions must coexist compatibly, or detect when they cannot do so. Extensions are in some ways "selfish" code: they want to have unfettered access to the internals of the mainline code or other extensions, and yet want to protect their own code from that same unfettered access. Since we intend extensions to themselves be extensible, we need a design that by default leaves them as extensible as the underlying browser, but that lets them (and the browser) protect their critical internals.

2.2 A hybrid approach

Flexibility and stability are not necessarily mutually exclusive, but they do require some additional information from the extensions. In particular, we might let extensions either declare explicit extension points of their own, or declare portions of themselves "frozen" and immune to further extension. We argue for the latter approach and its greater flexibility, and presents some preliminary steps in this direction.

3. Contrasting two extension models

Both Firefox and Chrome let authors extend two fundamental resources, namely the UI of the browser and the functionality of the browser, to varying levels and in different ways. These two resources are very different—one is declarative, one imperative and free-form—so both browsers define separate mechanisms for extending each. We compare each browser's approach to extending each resource, and give examples of actual problems that extensions face in each browser.

3.1 Extending the user interface

Chrome: Chrome deliberately aims to minimize the user interface (the "chrome") in its browser, and so offers a limited selection of interface elements to extensions: Chrome extensions use 16 fields to describe themselves, and of those only four are UI extension points. This is a "pull" model: Chrome collates extensions' self-descriptions, and creates the new UI elements (e.g. toolbar buttons) accordingly. Because extending the UI is intentionally limited, extensions cannot extend each other's UI.

[3] All extensions mentioned here are available from `http://addons.mozilla.org`.

StumbleUpon: Working around limitations[4] The four UI extension points supported in Chrome intentionally do not include a facility for creating toolbars. Some extensions, such as StumbleUpon, rely on a custom toolbar as their exclusive user interface. For these extensions, the only solution is to inject HTML into pages that mimics the appearance of a toolbar positioned at the top edge of the content area. Unfortunately, this workaround cannot be fully robust: vertical scrollbars will reveal that the "toolbar" is part of the page, CSS in the page may inadvertently restyle the toolbar, frame-busting code may be triggered to remove the toolbar, and the toolbar cannot be injected until after the page has finished loading (yielding a visually-jarring flicker). StumbleUpon lists these known issues, unsolvable without a richer extension API.

Firefox: By contrast, Firefox defines its entire UI using XUL, a markup language much like HTML, and CSS. It then exposes *everything* in its user interface to extensions via *overlays*, a reflective mechanism that lets extension authors "patch" XUL documents at runtime. Given a base document, an overlay can select nodes in the document by their `id` attribute and define new content to be inserted into them. Since nearly every element in Firefox's UI has an id, this allows developers to freely target the whole UI and to insert any UI element they choose. This is a "push" model: Firefox does not decide *a priori* which UI elements to expose, but rather extensions force themselves into their targets.

The overlay mechanism suffers from several important flaws. First, its current implementation has no well-defined semantics and exposes race conditions: the loading order of multiple overlays determines the document order of their composite result. This might change the event handling order of keyboard shortcuts, or cause UI elements to be pushed outside the visible space of the window. Second, nothing prevents multiple extensions from using overlays to create elements with duplicate identifiers, breaking the crucial uniqueness property of identifiers and potentially fooling other extensions into overlaying the wrong portion of the UI. Third, no practical way exists for an extension (or Firefox) to declare some document nodes as frozen for overlaying. Finally, no error is raised if an overlay *fails* to find a target element to extend, which masks errors among different versions of Firefox.

3.2 Extending the functionality

The program logic for both Firefox and Chrome extensions is written in JavaScript, but beyond that the two extension approaches differ.

Firefox: As with its UI, Firefox defines much of its functionality in JS and permits extensions to arbitrarily modify that code. Firefox extensions typically use two idioms, *wrapping* and *monkeypatching*, to inject themselves into existing functionality. These idioms use a combination of JS quirks, `eval`, and runtime rebinding of variables to change existing code's behavior. Since all chrome code (from Firefox and extensions alike) lives in a common namespace, it is frequently impossible to prevent one extension from modifying another's code. Said another way, there is no notion of extensions as distinct security principals, and hence extensions (and Firefox too) cannot ensure their own integrity at runtime.

AdBlock Plus and NoScript: A cautionary tale[5] AdBlock Plus and NoScript are two extremely popular Firefox extensions that between them block content from user-selected ad providers and script from all but user-whitelisted sites. AdBlock collaborates with several authors who maintain subscription lists of ad domains to block; these subscriptions update without user intervention. However, NoScript's development is supported by first-run ads that display each time an updated version of the extension is installed. For the many users running both extensions, this posed a problem for NoScript. In May 2009 a brief "arms race" began between the two extensions. NoScript used a known bug in AdBlock to hide its ads from detection; AdBlock asked for NoScript's domains to be added to the most common subscription list. NoScript countered by obfuscating the sources of its ads; the list was updated to match, and so on several times a day for most of a week. Eventually the filters were so over-broad as to break legitimate script on NoScript's installation page. Consequently, NoScript released an update that modified some internal functions of AdBlock to permanently construct a whitelist for NoScript's ads; that modification destabilized and broke AdBlock on legitimate sites.

While this particular spat was resolved successfully and without harm to users' machines or data, the implications are chilling. The same ability to amicably extend another extension can be abused to intentionally disable, cripple or subvert other extensions.

Chrome: By contrast, Chrome focuses heavily on robustness: extensions should not be able to subvert or destabilize the user's browsing session. To achieve this, Chrome extensions are segregated into individual security principals (unique origins, separated by the same-origin policy), and are further split into three layers: 1) JavaScript running in the context of the web pages being viewed, 2) script and HTML running in a separate process, and 3) binary components running in another separate process. Each layer can communicate with the next via message passing. Only the topmost layer can manipulate web pages; only the middle layer can coordinate the extension across multiple pages or access network resources, and even then it is constrained only to resources declared in the extension manifest. (The third layer is irrelevant for our purposes.)

One strong advantage of the Chrome approach is that each extension runs in its own JS namespace: extensions cannot

[4] `http://www.stumbleupon.com/sublog/su_chrome_extension/`

[5] `http://hackademix.net/2009/05/04/dear-adblock-plus-and-noscript-users-dear-mozilla-community/`, `http://adblockplus.org/blog/attention-noscript-users`

accidentally interfere with each other's code. No wrapping or monkeypatching is necessary—or possible. The layered architecture is both a strength and weakness: it helps ensure stability and prevent capability leaks, but is perhaps too limiting in preventing extensions from collaborating with each other: extensions cannot deliberately interfere with another's code, either. For instance, it would be impossible to implement a generic messaging client extension and later write supplemental extensions supporting specific protocols.

4. Proposed Language Support

We suggest that neither the free-wheeling sharing of Firefox nor the overly-partitioned approach of Chrome are appropriate designs for browser extensions. The former provides no aid in determining or ensuring extension compatibility, while the latter is too limiting for the compelling but unanticipated extensions that have been developed.

An idealized extensible browser permits extensions to modify the state, functionality, and appearance of the browser in a fine-grained, pervasive manner. Interactions between extensions are possible: extensions may communicate, may adapt to each other's presence, and may modify the mainline browser in benign ways. Extensions must have a mechanism for controlling their composition to resolve conflicts, either automatically or through user intervention.

Defining the notion of extension compatibility must account for the inherent differences between declarative UI extension and imperative full-blown code extension. Perhaps unsurprisingly, the former will be much easier. For both facets of extension, we focus on *commutativity* as a first-order approximation for extension independence: if two extensions "produce the same effect" on the browser regardless of their execution order, we might reasonably conclude they are independent of, and thus compatible with, each other. We expect many unrelated extensions to commute with each other. But when one extension relies on the presence or absence of another, commutativity is insufficient.

4.1 UI extension via semantic overlays

As a reasonable minimum requirement, we must ensure that the net UI effect of a set of installed extensions is defined solely in terms of the elements of the set: it should not depend on installation order, or naming conventions, or anything external to the extensions themselves. Therefore extensions must provide sufficient information to specify uniquely their loading order up to commutativity: for each extension, which other extensions must precede or follow it, and which other extensions do not matter?

Figure 1 presents an abstract language for defining overlays onto HTML documents. Everything above the divider is in extension authors' control; everything below is not. An overlay o defines a list of targets (either identifiers or the whole document) and content to append to them. Unlike Firefox overlays, ours must succeed or fail atomically: either all components of an overlay match the document or none do. We then give the programmer two additional constructions: *guards* (g) that specify additional properties that must hold when the overlay is being applied, and *composition operators* (c) that specify ordering constraints or optional portions of the overlay. Requirements (r) specify targets in the document; at minimum they include node identifiers. Guards are used to freeze portions of a document from further composition, or to request that they be pristine when a composition is applied. This capability is lacking in Firefox's system.

A *document* is a base document followed by an ordered sequence of compositions. We abstract the *state* of the document as four lists describing which requirements 1) must be defined in the document, 2) must be undefined, 3) have not yet been overlaid by a composition, or 4) must never again be overlaid, corresponding to the four guard types g. This machinery lets us define compositions c as *document transformers* that relate an initial document state before c is applied to a final state afterward, assuming the application succeeds. Finally, we can define when two overlays commute: if the output state of overlay o_1 cannot satisfy the needed input state of overlay o_2, then o_2 must precede o_1 and they do not commute. Any overlays not related by (chains of) such dependencies commute with each other. Such dependencies can then be used to find a valid loading sequence, if one exists, or find a set of conflicting extensions, if one does not.

Note that extension authors can modularly specify their compositions without knowing about any other extensions. Further, the compatibility analysis can be done by the browser without input from the user, and can simply inform the user (or developer) whether installing a new extension will yield a compatible composition or not. Thus with minimal developer overhead and no user effort, and without restricting UI extension points, we can solve the problems with Firefox overlays described above.

4.2 Code extension via aspects

The dependencies among overlays described above are insufficient to capture script conflicts. Indeed, expressing static dependencies between scripts is harder for two reasons.

Extensions change behavior by overwriting existing code to call new functions in the extensions instead. Currently this is done by redefining top-level functions or rewriting code using `eval`, both of which are hard to analyze and have semantic problems. In a concurrent conference-paper submission, we designed an aspect-oriented system for JavaScript where these code-injection points are made declarative and explicit. While the focus of that work was entirely on designing and implementing the aspect primitives, we consider a key benefit here: if all extension points used by extensions are declaratively specified, then it is much easier to see if two extensions conflict by seeing if their extension points overlap.

Unfortunately, since JS is a Turing-complete language, extensions' code cannot easily be analyzed for conflict because whether extension points *alias* can depend on arbitrary run-time behavior. Moreover, pairwise analyses may not be correct for larger sets of extensions. For example, suppose the baseline browser has three distinct functions bound to `f`, `g` and `h` and consider the following three extensions:

1. *At some point, extend* `f` *to always return 42.*
2. *At some point, extend* `g` *to always return 53.*
3. *Extend* `h` *to set* `f = g`.

Any two of these extensions are compatible with each other, as they advise different functions. However, if all three extensions are installed together, if extension 3 loads first then extensions 1 and 2 *might* conflict, depending on whether `h` happens to execute before the other extensions install their advice. Thus precisely detecting this type of conflict is undecidable statically.

Consequently, the only program with perfect information about installed extensions is the user's browser itself. Only it can correctly analyze running extensions and raise warnings if, when weaving advice into a function, the advice bodies do not commute: some alternate weaving order might yield different overall program execution. This dynamic approach will certainly yield more detailed information than is currently available, because it can track which extensions installed the advice, and so be of use to extension developers. However it may not be very helpful to end users, as the only actions available to them at runtime are either to abort the offending extensions or permit an unanticipated action, neither of which may be palatable.

Far better would be to detect statically, i.e., at extension installation time or earlier, whether several extensions conflict. The lazy weaving race scenario above is admittedly far-fetched: perhaps an unsound analysis could assume such strange code isn't used, and pragmatically detect valid weaving errors. Additionally, Douence et al. [3, 4] have made progress in analyzing aspects without reference to a base system. They define notions of strong and weak compatibility among aspects, which may be enough in practice to make an approximate static dependency analysis feasible, or to decide exactly what runtime checks are needed for a sound analysis.

Finally, as with overlays, extensions may want to "freeze" some of their functionality against modification by other extensions. We are not the first to notice this; Aldrich [1] proposed the notion of "open modules" to achieve such freezing, and the technique should fit well with our own aspect work and the browser setting.

4.3 Enforcing security policies

We have said nothing yet about enforcing security policies on these well-composed extensions. Instead, we see all of the challenges above for simply defining a clearer semantics for extensions and compatibility as being prerequisites to security efforts.

Chrome's extension security model takes a capability-based approach, wherein extensions declare which resources (e.g., "browser history", "network access to `foo.com`") they require and users grant those permissions at extension install-time. We do not yet address such policies, except to note that when extensions may interact freely (unlike in Chrome), more work must be done to prevent a confused-deputy attack or outright collusion between extensions; this extends the threat model in [2]. Additional care must be taken to protect such a composite system from runaway extensions, as in [5].

$$
\begin{array}{rrcll}
o \in & \textsc{Over} & ::= & \cdot \mid (r, h), o \mid (\texttt{\#Doc}, h), o & \\
g \in & \textsc{Guard} & ::= & o & \\
 & & \mid & \mathit{Require}\ r\ g & \text{–}\textit{g requires r be defined} \\
 & & \mid & \mathit{Reject}\ r\ g & \text{–}\textit{g requires r be undefined} \\
 & & \mid & \mathit{First}\ r\ g & \text{–}\textit{r must not be overlaid before g} \\
 & & \mid & \mathit{Last}\ r\ g & \text{–}\textit{r must not be overlaid after g} \\
c \in & \textsc{Comp} & ::= & g \mid c\,;c & \text{–}\textit{composition} \\
 & & \mid & c? & \text{–}\textit{optional sequencing} \\
r \in & \textsc{Req} & ::= & i \mid \cdots & \text{–}\textit{ids or other requirements} \\
\hline
S \in & \textsc{State} & ::= & \{\mathit{Def} : \vec{r}, & \text{–}\textit{currently defined} \\
 & & & \quad \mathit{Undef} : \vec{r}, & \text{–}\textit{currently undefined} \\
 & & & \quad \mathit{Clean} : \vec{r}, & \text{–}\textit{has not yet been overlaid} \\
 & & & \quad \mathit{Frozen} : \vec{r}\} & \text{–}\textit{may never again be overlaid} \\
d \in & \textsc{Doc} & ::= & h \mid d[c] & \text{–}\textit{overlay sequence} \\
i \in & \textsc{Ident}, & h \in & \textsc{Html} &
\end{array}
$$

Figure 1. Abstract overlays with freezing and composition

5. Conclusions

We have described the space of web-browser extensions, and argued that they are in need of programming-language research. We have sketched the essential features of two browser extension systems, and identified key strengths and flaws of each. We proposed two language mechanisms for improving the development of extensions, and suggest that together they provide a more compelling platform for supporting extensions.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, 2005.

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

[3] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Generative Programming and Component Engineering*, 2002.

[4] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Aspect-Oriented Software Development*, 2004.

[5] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or revenge of the son of the Lisp machine). In *International Conference on Functional Programming*, 1999.