Detecting Conflicts Among Declarative UI Extensions

Benjamin S. Lerner
Brown University
blerner@cs.brown.edu

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

Abstract

We examine *overlays*, a flexible aspect-like mechanism for third-party declarative extensions of declarative UIs. Overlays can be defined for any markup language and permit extensions to define new content that is dynamically woven into a base UI document. While powerful, overlays are inherently non-modular and may conflict with each other, by defining duplicate or contradictory UI components. We construct an abstract language to capture core overlay semantics, and design an automatic analysis to detect inter-extension conflicts. We apply the analysis to a case study of Firefox extensions, finding several real-world bugs. Our analysis provides low-level feedback to extension developers and high-level reports to end users. Finally, we show how variants of overlays more expressive than those of Firefox complicate conflict detection.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques — User interfaces; D.3.2 [*Programming Languages*]: Language Classifications — Design languages

General Terms Languages, Experimentation

Keywords Extensions, web browsers, overlays, conflicts

1. Introduction

Many modern markup languages now exist for designing user interfaces—e.g., Mozilla's XUL, Microsoft's XAML, and HTML—and their distinguishing feature is providing a *declarative* means of specifying the structure of a UI separately from its behavior. The use of such languages is growing rapidly: all webapps, classic applications (like Firefox), and even entire smartphone OS shells (such as HP's webOS or Mozilla's Boot2Gecko) are written in this manner.

Mozilla exploited this declarative structure in a novel way: they created a new mechanism known as *overlays* that allow developers to define UI portions in separate XUL documents that are then dynamically woven together into a complete document. This concept is not XUL-specific, and can be applied to any markup language. Overlays (informally) consist of a *selection* of some element in the mainline document and a *content subtree* to be inserted into that element. The terminology is deliberately suggestive: overlays are akin both to "tree-shaped patches" and to aspects. Gecko (Mozilla's rendering engine) then merges the overlays dynamically. Figure 1 shows a "Hello world" example written in XUL, a simple overlay targeting it, and its composition with the base document. Applications like Firefox use this ability heavily to modularize their UI definitions into many smaller documents.

Mozilla applications expose the overlay mechanism to third parties and thereby enable a uniquely powerful extension mechanism. Such third party extensions can enhance or modify the program's base functionality in arbitrary ways; overlays are used to integrate the extension's UI into the existing UI. Moreover, end users can freely install extensions to customize their browser however they wish. This expressiveness has led to the widespread popularity of Firefox extensions—hundreds of millions of users have downloaded extensions billions of times [13].

1.1 Challenges of UI Extension via Overlays

Much like aspects, overlays shift the challenge from coding around insufficiently-expressive hooks to reasoning about overly powerful ones. Most UI frameworks let widget authors completely control their UI and the API by which others use the widget. While convenient for widget authors, a developer who requires functionality from an existing widget that is not provided is helpless to remedy that lack.

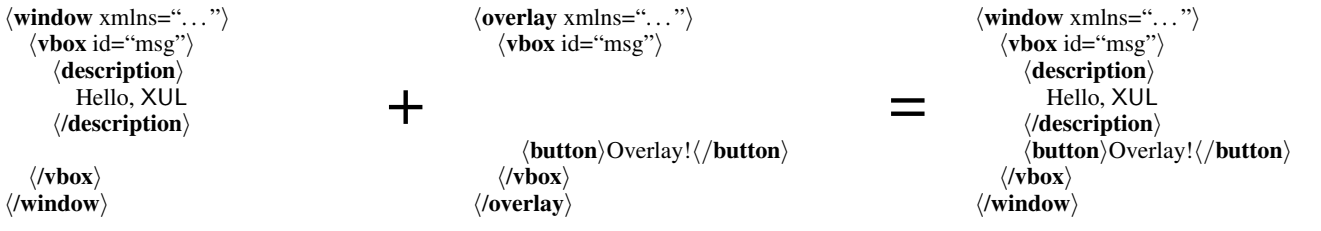
By contrast, overlays permit fine-grained third-party integration into existing UIs *without any cooperation* from the underlying UI. As with aspects, this flexibility cuts both ways. Extension authors can precisely modify nearly any facet of the underlying application, and avoid the inflexibility of a predetermined set of extension points. However, extensions might in turn be extended without their consent, so they, like the UI they target, cannot protect their own integrity. This challenge is made worse since every user might have installed a different browser version and a unique combination of extensions. Unwanted or unforeseen extension interactions may cause buggy behavior at runtime, but it is simply infeasible for developers to anticipate all possible interactions.

The sliver lining, however, is that markup languages are *not* general-purpose, and overlays are *not* as expressive as traditional aspects. Accordingly, the challenge of detecting overlay-composition conflicts is tractable, though as we will show, even reasonably modest expressiveness enhancements will greatly complicate the conflict detection algorithm.

1.2 Contributions

- We identify three broad groups of overlay composition errors: structural violations of overlay requirements, and structural or semantic violations of the underlying markup language.
- We construct an abstract notion of overlays, then show how to summarize their effects as *document transformers*. These transformers encode the semantics of the markup and overlays in a language-neutral manner.
- We develop an automatic analysis to detect the three overlay incompatibilities above. Our analysis uses the transformers

DLS'12, October 22, 2012, Tucson, Arizona, USA.



(a) "Hello, XUL" base document (b) Overlay targeting the ⟨**vbox**/⟩ (c) Composition of base with overlay

Figure 1: Simple example of XUL, overlay, and composite result (overlay is appended to existing content)

above to compute a dependency graph among the overlays. This analysis can detect errors at extension-installation time or earlier, preventing the end user from experiencing broken UIs. We detect a dozen conflicts among the top 350 extensions for Firefox 3.0.

- We examine more general combinators for overlay composition, and show that seemingly-modest extra expressiveness makes conflict checking exponentially harder.

The rest of this paper is structured as follows. Section 2 presents a brief tutorial of overlays as implemented in Firefox, and Section 3 presents the three forms of inter-extension conflict that we target. Section 4 develops our framework for modeling overlays and detecting conflicts among them. Section 5 presents a case study applying this approach to Firefox extensions. Section 6 motivates two enhancements to our modeling language, and explains why such enhancements greatly complicate conflict checking. Finally, Section 7 presents related and future work.

2. A tutorial on Firefox-style overlays

We introduce overlays by a simple example, and describe the steps involved in applying overlays to a base document. Conceptually, overlays behave like "patches on trees": they select a target location in the tree and define new content to be inserted at that location.

Identifying targets: Consider the base document shown in Fig. 1a, and the simple overlay shown in Fig. 1b. An overlay defines one or more actions, denoted by the children of the ⟨**overlay**/⟩ root node. These children indicate a *target*, namely those nodes in the base document with the same tag name and id: here, (**vbox**, `"msg"`).

Inserting content: Once an action's target is identified, the action's children are cloned and appended to the target node.[1] These inserted nodes are now part of the base document; subsequent extensions may overlay them as if they had been present all along. Additionally, any attributes on the action are cloned onto the target, overwriting any that are already present.[2]

Failing softly: By design, extensions may apply to multiple programs (e.g., the AdBlock Plus extension applies to both Firefox and Thunderbird), and several extensions include optional code that should be used if other extensions are present (e.g., TabMix Plus detects the presence of Session Manager and modifies its UI accordingly). Gecko supports such optional code by silently ignoring actions that do not match any targets. This can cause problems; we address optional code differently, below.

[1] We ignore for now when to run ⟨**script**/⟩s that are cloned in this process.

[2] Firefox overlays, like textual patches, can also *remove* existing text; this ability is rarely used, and we do not support it here.

3. Kinds of overlay conflicts

The simple operational definition of overlay application in the previous section is designed for resilience: for any base document and any set of input overlays, it will produce a composite document. However this deliberately masks three kinds of conflicts, which we wish to detect and help correct: violations of document well-formedness, language-specific semantic violations, and load-order violations. For brevity (and generality), we will abandon the concrete syntax of Fig. 1b, and instead write overlays using an abstract syntax that we define more precisely in Section 4.1.

3.1 Structural uniqueness constraints: node ids

Consider the following simple document snippet:
⟨**p** id="greeting"⟩⟨**span**⟩Hello,⟨/**span**⟩⟨/**p**⟩
Suppose two extension authors wanted to complete the greeting by supplying the subject:

OV1: *Overlay*(*Insert*(`p#greeting`, end, ⟨**span** id="subj"⟩stranger.⟨/**span**⟩))

OV2: *Overlay*(*Insert*(`p#greeting`, end, ⟨**span** id="subj"⟩friend.⟨/**span**⟩))

(Read OV1 as declaring, "*Insert*, into the node matching the selector `p#greeting`, at the *end* of its existing content, the new node ⟨***span*** *id="subj"*⟩*stranger.*⟨/***span***⟩.") And suppose a later extension author wanted to modify the greeting:

OV3: *Overlay*(*Insert*(`p#greeting`, end, ⟨**span** id="mod"⟩and good day,⟨/**span**⟩))

Two problems arise if a user applied both OV1 and OV2: one in the structure of the composite, and one in its English-level content. Structurally, the composite document is not well-formed, because two ⟨**span**/⟩ elements exist with the same id "subj". An automated conflict-detection system can and should flag such errors. By contrast, the English content of the sentence—"Hello, stranger. friend." or "Hello, friend. stranger.", depending on the order of insertion—is nonsensical, but this is beyond the scope of overlay conflict detection. To detect and report structural conflicts, we must have a way of summarizing the effect of each extension, particularly any well-formedness constraints (e.g., element ids must be unique), and check all pairs of effects for overlap.

On the other hand, if the user applied OV1 and OV3 (or OV2 and OV3), the result would be a well-formed HTML document, since no duplicate ids would be present. Note that the sentence might still be nonsensical English—"Hello, stranger. and good day,"—due to insertion order. However, the extensions are *not* in conflict, but merely imprecise: they only claim to be inserted at the end of existing content, and make no mention of their order relative to other extensions. We will return to errors of this sort in Section 3.3.

3.2 Semantic uniqueness constraints: hotkey bindings

Uniqueness of identifiers is essentially a language-agnostic property (all markup-based languages share this kind of well-formedness

constraint), but other properties depend on the semantics of the markup language being used.

For instance, XUL allows developers to declare application-wide hotkeys using ⟨**key**/⟩ elements. Assume the base document defines a container tag ⟨**keyset** id="keys"/⟩. Then define three extensions:

OV4: *Overlay*(*Insert*(`keyset#keys`, end,
⟨**key** key="F" oncommand="alert("1")"/⟩))

OV5: *Overlay*(*Insert*(`keyset#keys`, end,
⟨**key** key="F" oncommand="alert("2")"/⟩))

OV6: *Overlay*(*Insert*(`keyset#keys`, end,
⟨**key** key="G" oncommand="alert("3")"/⟩))

The composite document is well-formed, but the intended semantics of application-wide hotkeys are not satisfied: a hotkey can only trigger one unique action, and both OV4 and OV5 expect to run in response to the key "F". A conflict-detection system should detect that OV4 and OV5 conflict with each other, but that neither one conflicts with OV6.

If the only potential conflicts were global uniqueness of resources, such as ids or hotkeys, the conflict-detection problem would be straightforward. However some conflicts are more local in scope. Consider the semantics of XUL's ⟨**radiogroup**/⟩s, where at most one of its several ⟨**radio**/⟩ button children are selected.

OV7: *Overlay*(*Insert*(`radiogroup#g1`, end,
⟨**radio** selected="true"⟩G1:ov7⟨/**radio**⟩))

OV8: *Overlay*(*Insert*(`radiogroup#g1`, end,
⟨**radio** selected="true"⟩G1:ov8⟨/**radio**⟩))

OV9: *Overlay*(*Insert*(`radiogroup#g2`, end,
⟨**radio** selected="true"⟩G2:ov9⟨/**radio**⟩))

Here, OV7 and OV8 conflict with each other, as both attempt to add a selected item to the same group, but neither conflicts with OV9, which adds a selected item to a different group.

The equivalent problem in HTML is still more challenging, as there is even less structural information available. Radio ⟨**input**/⟩s are grouped by their name attribute, rather than by a common ⟨**radiogroup**/⟩ ancestor, and so multiple groups could be interleaved in the markup. Thus any automated conflict detector must be flexible enough not to assume a particular hierarchical tree structure, but respond to attributes and other properties in the tree as well.

3.3 Load-order constraints

We saw one example of load-order dependencies already, where OV3 needed to assert that it loaded before OV1 or OV2, but had no way of doing so. In a less-contrived example, extensions can build upon each other in useful ways. Firefox extensions commonly add new submenus to the "Tools" menu: for example, the Session Manager extension adds a submenu with menu items to save currently open tabs as a "session", to load a previously-saved session, and to delete saved sessions:

OV10: *Overlay*(
Insert(`menu#tools-menu`, end,
⟨**submenu** id="sessionManagerMenu"⟩
⟨**item**⟩Load session…⟨/**item**⟩
⟨**item**⟩Save session…⟨/**item**⟩
⟨**item**⟩Delete session…⟨/**item**⟩
⟨/**submenu**⟩))

A common use-case for Session Manager is to use sessions as "temporary bookmarks", where the session is used once and then discarded. A second extension might make this available as its own menu item:

$$
\begin{array}{rrll}
c \in & \mathit{Comp} & ::= & g & \textit{–guarded overlay} \\
 & & \mid & c\,;c & \textit{–sequential composition} \\
 & & \mid & c\,!\,c & \textit{–exclusive composition} \\
 & & \mid & c? & \textit{–optional composition} \\
g \in & \mathit{Guard} & ::= & o & \textit{–overlay} \\
 & & \mid & \mathit{Require}(\vec{r}, g) & \textit{–must be defined} \\
 & & \mid & \mathit{Forbid}(\vec{r}, g) & \textit{–must be undefined} \\
 & & \mid & \mathit{First}(\vec{r}, g) & \textit{–must not be overlaid yet} \\
 & & \mid & \mathit{Last}(\vec{r}, g) & \textit{–must not be overlaid again} \\
o \in & \mathit{Overlay} & ::= & \mathit{Overlay}(\vec{a}) & \\
a \in & \mathit{Action} & ::= & \mathit{Insert}(s, w, \mathit{htmlSubtree}) & \\
 & & \mid & \mathit{Modify}(s, \vec{t}) & \\
r \in & \mathit{Resource} & = & \mathit{Sel}(s) \uplus \mathit{Id}(\mathit{identifier}) \uplus & \\
 & & & \mathit{Key}(\mathit{keyName}) \uplus & \\
 & & & \mathit{Selected}(\mathit{htmlSubtree}) \uplus \cdots & \\
s \in & \mathit{Sel} & ::= & \mathit{tagName}\#\mathit{identifier} & \\
w \in & \mathit{Where} & ::= & \text{before} \mid \text{after} \mid \text{start} \mid \text{end} & \\
t \in & \mathit{Attribs} & ::= & (\mathit{attribName}, \texttt{attribValue}) &
\end{array}
$$

Figure 2: Abstract syntax for overlays

- **OV11**: *Overlay*(*Insert*(
`submenu#sessionManagerMenu`, start,
⟨**menu**⟩Load/discard session…⟨/**menu**⟩))

These two extensions are not in conflict, but OV11 *depends upon* OV10 being loaded first: otherwise, OV11's target node does not yet exist. The current overlay loader in Gecko does not ensure any deterministic loading order; indeed, individual actions of a single overlay may be applied in arbitrary order relative to each other and to actions from other overlays. Code such as OV11 seems to work in practice, but there is no guarantee it will continue to do so.

4. Abstract overlays and document transformers

Overlays present a compelling means for modularizing UI definitions, but specific implementation details make reasoning about them challenging. In this section and the next, we define an analysis to compute a compatible load-order among a set of overlays, or else to find and report a set of incompatible overlays. Our strategy is to: 1. define an *abstract language* of overlays and combinators (sequential, optional, or one-of-several composition), 2. interpret each overlay as a *document transformer* describing its requirements and effects, 3. compute a *conflict dependency graph* among the transformers, and 4. model the base document and then topologically *sort* the graph, or else find a cycle. Supporting optional (or one-of-several) compositions is provably hard, so finally we must 5. use heuristics to handle optional compositions. These steps will be described in Section 4.1 through Section 4.5.

4.1 An abstract overlay language

Figure 2 defines the syntax of our abstract overlay language; we have shown simple examples of it in the previous section. In this section we explain the semantics of the language; in Section 5.1 we highlight some of the accommodations needed to map concrete XUL overlays to this abstraction.

4.1.1 Overlays, resources and guards

An *Overlay* consists of a mandatory, atomic set of actions: all actions must apply successfully in sequence, or else the entire overlay fails to apply. Each action selects a target s and can either

Insert new content into it or *Modify* existing attributes. Insertions occur *before* or *after* the target node, or at the *start* or *end* of its contents.

The examples in the preceding sections highlight several distinct types of conflicts that must all be properly accounted for by a conflict detection algorithm. The hotkey example requires knowing that each key must be unique within a document. The options list example must include the HTML semantics that at most one option be selected within a group. And the hello-world example shows that even if all uniqueness constraints are known and modeled, an analysis may potentially still miss higher-level conflicts that are not expressible within the markup language. To support all of these kinds of conflicts in a single system, in a uniform, declarative way, our analysis will be defined in terms of an abstract set of *resources*, which may include pieces of the overlay itself (e.g., ids, keys, selectors), and *guards*, which may be automatically inferred or may be assertions manually provided by the overlay author to help further constrain when the overlay applies successfully. Finally, our language includes *compositions* to express the variations needed for targeting multiple applications. We illustrate their behavior by revisiting the earlier examples.

"Hello, world", revisited: Recall that OV3's English-level conflict with OV1 and OV2 occurs since OV3 has no way to assert it must load before the others. The authors of OV1 or OV2 might defensively try to accommodate OV3 by claiming to overlay OV3's content but in fact do nothing to it:

OV1*: *Overlay*(
Insert(`p#greeting`, end,
⟨**span** id="subj"⟩stranger.⟨/**span**⟩),
Modify(`span#mod`))

(In this paper, overlays marked with a star are broken variations of a correct overlay; those marked with an apostrophe are working, alternate versions.) This *Modify* action has the effect of making OV1* depend upon—and hence load after—OV3, but there are three fundamental problems with this approach. First, OV1 does not actually modify the ⟨**span**/⟩ produced by OV3, so OV1* should not claim to modify it. Second, this approach is non-local: OV1 and OV2 were written first and could not have known about OV3, so it must be OV3's responsibility to impose ordering constraints on itself, rather than expect others to accommodate it. Third, and most important, OV1* will not successfully apply if OV3 is not present, a behavior that differs from OV1.

Instead, a better approach is to assert extra *guards* in the definition of OV3, so that it can express the constraint that it must not apply when OV1 or OV2 is present:

OV3': *Forbid*(*Id*(`subj`),
Overlay(*Insert*(`p#greeting`, end,
⟨**span** id="mod"⟩
and good day,
⟨/**span**⟩)))

Read this as "OV3' applies successfully to documents in which the *Overlay* successfully applies, but not those documents that match the *Id* `subj`". The four types of guarded compositions apply successfully when:

- *Require*($\vec{r}$, g): When g succeeds and all resources in r are present in the document
- *Forbid*($\vec{r}$, g): When g succeeds and all resources in r are not present in the document
- *First*($\vec{r}$, g): When g succeeds and no resources in r have been overlaid by some prior overlay
- *Last*($\vec{r}$, g): When g succeeds and no resources in r will be overlaid by some future overlay

Hotkey responses, revisited: Rather than hard-code knowledge of the markup language's semantics into the conflict-resolution algorithm, constraints can be encoded using additional guards. For example, OV4 and OV5 may each assert they must be the *last* key to use the letter "F":

OV4': *Last*(*Key*(F), *Overlay*(
Insert(`keyset#keys`, end,
⟨**key** key="F" oncommand="…"/⟩)))
OV5': *Last*(*Key*(F), *Overlay*(
Insert(`keyset#keys`, end,
⟨**key** key="F" oncommand="…"/⟩)))

Now the presence of both OV4' and OV5' will trigger a conflict due to these general-purpose *Last* guards, rather than due to an algorithm finely tuned to XUL's idiosyncrasies.

Rather than require overlay authors to write these *Last* guards every time they use a ⟨**key**/⟩ element, we may mechanically derive them from the overlay markup. In particular, these language-specific details can be encoded during the parsing of concrete-syntax overlays into the abstract overlay language. This leaves the underlying conflict-detection algorithm agnostic to the initial input language, but also relieves authors from the tedium of writing "obvious" guards.

Handling ⟨script/⟩ tags: Script tags violate the static, purely declarative nature of markup. In both HTML and XUL, scripts are (by default) run as they are encountered, and can affect how the remainder of the input file is parsed. Overlays intrinsically require parsing both the base document and the overlay, so there are several possible choices for when an overlay mechanism should execute scripts. We treat scripts as inert within the overlay document, and execute them as they are cloned into the target document. Overlays apply to the base document after it has finished parsing, and therefore after its scripts have executed.

4.1.2 Composing overlays within one extension

So far, (guarded) overlays are "all-or-nothing": they either apply successfully or not at all. Extension authors may want to control how their overlays are applied in a more fine-grained fashion. To support such intentions, our overlay language includes three types of composition: sequencing, one-of-several targeting, and optional components.

Sequencing helps modularize overlay code: the extension author can separate overlays into logically-distinct pieces, and ensure that they are applied to the base document in the desired order. The sequential composition $c_1 \,;\, c_2$ succeeds for documents D whenever c_1 succeeds in D and c_2 succeeds in the document resulting from applying c_1 to D.

One-of-several targeting is useful for the relatively frequent case where extensions are written that may apply to multiple, "similar" applications. For example, an extension may add a command to the "Tools" menu of both Firefox and Thunderbird. In Firefox, that menu is ⟨**menu** id="tools-menu"/⟩, but in Thunderbird it is ⟨**menu** id="tasksMenu"/⟩. Currently, extension authors essentially write

Overlay(*Insert*(`menu#tools-menu`, end, *new content*),
Insert(`menu#tasksMenu`, end, *new content*))

This implicitly relies on Gecko's quirky, silent dropping of individual overlay actions with missing targets. This often works in practice, but it is subtly wrong: if another extension happens to insert a ⟨**menu** id="tasksMenu"/⟩ element into Firefox, this extension would insert *new content* twice, with likely malformed results (e.g., if the inserted content contains elements with ids, these would

become duplicated). In our overlay language, the one-of-several composition allows extension authors to indicate *mutually exclusive overlays*. A composition $c_1 \,!\, c_2$ succeeds in a document D if either c_1 or c_2 (or both) can apply successfully to D, but will apply only one of them. We choose a left-biased implementation for one-of-several targeting (i.e., if c_1 can apply then it does so, and only if it fails will c_2 be tried), but an unbiased and non-deterministic implementation is equally feasible. Neither semantics can automatically detect, say, an extension that rudely adds `menu#tasksMenu` to Firefox; conventionally, this is resolved by using extension-specific prefixes for ids.

Finally, extensions often deliberately include optional behaviors that are installed only when other extensions are installed as well. Once again Firefox extensions rely on Gecko's silent dropping of unmatched overlays to implement these optional features, and this works successfully. The unfortunate consequence, though, is that Gecko cannot distinguish optional components (to be ignored) from typos (to be surfaced as errors). In our overlay language, components must be marked explicitly as $c?$ for the composition and conflict-detection algorithms to consider them optional; everything else must succeed or fail as a unit.

4.2 Overlays as document transformers

A precise conflict-detection algorithm must keep track of resources and must keep track of four kinds of guards on those resources. Moreover, as OV10 and OV11 show, it must record how those constraints and available resources *change* as a consequence of sequentially applying extensions' compositions overlays, so that later compositions can see the effects of preceding ones. Ultimately, it must compute, given a base document D and a set of extensions $\vec{e}$ that define compositions $\vec{c}$, some permutation of the extensions that yields a feasible loading order (i.e., the composition $D\,;c_{\pi(1)}\,;\cdots;\,c_{\pi(n)}$, for some permutation π, is valid according to the analysis), or else demonstrate some subset of the overlays that are conflicting.

To model how resources change as a consequence of a single overlay, the analysis will view an overlay as a *document transformer* that takes an input document and produces a modified document as a result; the analysis will then be concerned with tracking resources as they are manipulated within successive documents. To start, we model the state of the document by a record $\{Def, Undef, Clean, Frozen\}$ of sets of resources describing the set of requirements that must be defined, that must not be defined, that must have not yet been overlaid and that must never again be overlaid in order for the overlay to succeed. (For notational brevity, we elide empty sets in records, below.) These sets give the overlay a small footprint: nothing can be inferred about resources that are not mentioned. For instance, an empty input Def set does *not* assert that the input document is empty, but rather that nothing is known to be defined.

Our dependency analysis will reason about guarded overlays as a pair of states (S_i, S_o), which can be thought of as the weakest precondition and strongest postcondition of the overlay; an example is shown below. Computing this interface is purely structural (details are in the appendix, Fig. 3). We ignore for now the complications of one-of-several and optional compositions, and return to them in Section 4.5.

Of the eight sets in (S_i, S_o) for some overlay, the S_o^{Def} and S_o^{Undef} sets are uniquely determined from the overlay itself, S_i^{Def}, and S_i^{Undef}. Additionally, the S_i^{Frozen} and S_o^{Clean} sets are redundant: anything that was $Clean$ before the overlay applied, and that was unused by the overlay, will remain $Clean$; similarly, the overlay need only check the *output* $Frozen$ set of the preceding overlay, and not vice versa. The remaining four sets can be specified or extended independently, using the four guard types above.

This redundant representation has two advantages. First, it represents the effects of an overlay in a uniform way: we can compare the input and output states without needing to know any internal details of the overlay. Second, these states are closed under composition: a state pair can describe the effects of a sequence of overlays, again without needing to know the details of the sequencing. We rely on this fact heavily in the dependency analysis in the next section.

The interface of "Hello, world": Recall the definitions of OV1 and OV3':

OV1: $Overlay($*Insert*(`p#greeting`, end, ⟨**span** id="subj"⟩stranger.⟨/**span**⟩))

OV3': $Forbid(Id($`subj`$),$ $Overlay($*Insert*(`p#greeting`, end, ⟨**span** id="mod"⟩ and good day,⟨/**span**⟩))))

The meaning of OV1 in words is "for any document containing a node `p#greeting` and not containing nodes matching `span#subj` or any node with id `subj`, OV1 will produce a document that contains `p#greeting`, `span#subj` and a node with id `subj`". In symbols, this becomes the state pair (S_i^1, S_o^1):

$$S_i^1 = \left\{ \begin{array}{r} Def = \{Sel(\texttt{p\#greeting})\} \\ Undef = \{Sel(\texttt{span\#subj}), Id(\texttt{subj})\} \end{array} \right\}$$

$$S_o^1 = \left\{ \begin{array}{r} Def = \{Sel(\texttt{p\#greeting}), \\ Sel(\texttt{span\#subj}), Id(\texttt{subj})\} \end{array} \right\}$$

The Sel resources describe the structural changes to the document, so that overlays can detect if their targets exist. The Id resources encode the unique-id requirements.

Similarly, the effect of OV3' in words is "for any document containing a node matching `p#greeting` and not containing nodes matching `span#mod` or any node with id `mod` or any node with id `subj`, OV3' will produce a document that contains `p#greeting`, `span#mod` and a node with id `mod` and still does not contain any node with id `subj`". In symbols, this is the state pair (S_i^3, S_o^3):

$$S_i^3 = \left\{ \begin{array}{r} Def = \{Sel(\texttt{p\#greeting})\} \\ Undef = \{Sel(\texttt{span\#mod}), \\ Id(\texttt{mod}), Id(\texttt{subj})\} \end{array} \right\}$$

$$S_o^3 = \left\{ \begin{array}{r} Def = \{Sel(\texttt{p\#greeting}), \\ Sel(\texttt{span\#mod}), Id(\texttt{mod})\} \\ Undef = \{Id(\texttt{subj})\} \end{array} \right\}$$

The $Id($`subj`$)$ constraint is added to the input $Undef$ set by the $Forbid$ assertion that the specified resource not exist, and to the output $Undef$ set because OV3' does not itself cause that resource to be defined.

4.3 Determining the overlay conflict graph

Consider just the two overlays OV3' and OV1. Can OV3' be composed with OV1, and if so in what order? Suppose the system tried OV1 ; OV3', applying OV1 first. Then the *input* state of OV3' must be compatible with the *output* state of OV1. However, it is clear that $S_o^1.Def \cap S_i^3.Undef = \{Id(subj)\} \neq \emptyset$. In words, something that OV3' requires to be undefined is guaranteed to be defined by OV1. This one contradiction suffices to prohibit the ordering OV1 ; OV3'.

On the other hand, suppose the system tried the other order, OV3' ; OV1. This time, the intersection test above succeeds, as well as several others. In general, for OV3' to precede OV1 the system must check:

$$S_o^3.Def \cap S_i^1.Undef = \emptyset \quad (1)$$

$$S_o^3.Def \cap \text{defs}(\text{OV1}) = \emptyset \quad (2)$$

$$S_o^3.Frozen \cap \text{used}(\text{OV1}) = \emptyset \quad (3)$$

$$\text{reqs}(\text{OV3'}) \cap S_i^1.Clean = \emptyset \quad (4)$$

These equations assert that OV3' must not define anything OV1 requires as undefined, nor anything that OV1 itself defines; it also must not freeze anything that OV1 later uses, or require anything that OV1 asserts to be clean. These four equations do hold for OV3' and OV1, and hence OV3' ; OV1 is a valid composition order for the two overlays.

The resulting composition *itself* can be represented by a state-pair interface $(S_i^{3,1}, S_o^{3,1})$ that is computable from S_i^1, S_i^3, S_o^1 and S_o^3. The rule (shown in Fig. 4, in the appendix), checks the four equations above, and the resulting interface should mean "in a document satisfying the combined requirements of OV3' and OV1, as described by $S_i^{3,1}$, the composition OV3' ; OV1 will result in a document satisfying their combined guarantees, as described by $S_o^{3,1}$". Consequently, the input state $S_i^{3,1}$ must be a combination of the Def and $Undef$ sets of S_i^3 with those of S_i^1. To do otherwise would effectively enforce that OV3' define everything needed by OV1, which is not the intended semantics:

$$S_i^{3,1} = \left\{ \begin{array}{r} Def = \{Sel(\texttt{p\#greeting})\} \\ Undef = \{Sel(\texttt{span\#mod}), Id(\texttt{mod}), \\ Sel(\texttt{span\#subj}), Id(\texttt{subj})\} \end{array} \right\}$$

$$S_o^{3,1} = \left\{ \begin{array}{r} Def = \{Sel(\texttt{p\#greeting}), \\ Sel(\texttt{span\#subj}), Id(\texttt{subj}), \\ Sel(\texttt{span\#mod}), Id(\texttt{mod})\} \end{array} \right\}$$

In words, this says that "for any document containing a node matching `p#greeting` and not containing nodes matching `span#mod` or `span#subj` or any nodes with id `subj` or `mod`, the composition (OV3' ; OV1) will produce a document containing `p#greeting`, `span#mod`, `span#subj`, and nodes with id `subj` or `mod`".

The approach Eqs. (1) to (4) above define when one overlay may feasibly follow another. If, however, *any* equation is unsatisfied (as when applying OV1 before OV3'), the analysis knows such an ordering is infeasible: in the example above, OV3' *must not follow* OV1 because of Eq. (1) and $Id(\texttt{subj})$. The analysis records such observations in a directed graph, whose nodes are overlays, and whose edges are the pairwise must-not-follow relation. For diagnostic purposes later, it annotates the edges with *which* of the four equations above were unsatisfied. This is the *conflict graph* for a set of extensions.

Note that it is possible that extension X must-not-follow extension Y according to Eqs. (1) to (4), and Y must-not-follow extension Z, but that X may feasibly follow Z:

- X: $Overlay(Insert(\texttt{p\#w}, \langle \textbf{p} \text{ id="x"}/\rangle))$
- Y: $Overlay(Insert(\texttt{p\#x}, \langle \textbf{p} \text{ id="y"}/\rangle))$
- Z: $Overlay(Insert(\texttt{p\#y}, \langle \textbf{p} \text{ id="z"}/\rangle))$

X must-not-follow Y because Y uses something that X defines; similarly for Y and Z. But X and Z are independent of each other, and can be applied in either order. However, when all three extensions are loaded together, the conflict graph shows that Z transitively must-not-follow X because of a *path* of pairwise must-not-follow edges.

Recall that, given some base document D and a set of extensions $\vec{e}$, the analysis must compute either a feasible composition order, if one exists, or else a set of conflicting extensions. It can resolve both questions simply by computing whether the conflict graph is *acyclic*. If the graph contains a cycle, then there exist extensions $e_1, \ldots e_n$ such that e_i must not follow e_{i+1}, and e_n must not follow e_1. Transitively, each extension must not follow itself, which is problematic: there cannot exist a loading order for which all of these constraints are satisfied, and so extensions $e_1, \ldots, e_n$ are in conflict. Moreover, the annotations on each edge of the cycle explain to the user precisely *why* the extensions conflict.

4.4 Modeling the initial document

If the conflict graph is acyclic, then any topological sort of the graph will yield a loading order that respects all extension dependencies. To ensure they combine properly with a given base document, two further checks remain: first, does the base document in fact satisfy the input requirements of the resulting composition? Additionally, does the composition define anything already defined by the document? These two checks amount to precisely the same four conditions as for sequencing of two compositions.

To resolve these remaining questions, we simply model the base document itself by its interface, as if it too were an overlay. The interface for a base document begins with the empty input state, and produces an output state containing everything defined in the base document. Adding this interface as a node to the conflict graph obtains the needed sequencing checks "for free". The final algorithm for computing conflicts and loading order is:

1. For each extension E_i and its corresponding guarded overlay g_i, compute its interface I_i.
2. For the base document D, compute its interface $I_0 = I_D$.
3. Construct the conflict graph, with nodes I_i and I_0, and edges $I_i \rightarrow I_j$ if and only if I_i must-not-follow I_j using the four conflict rules above.
4. If the graph is acyclic, and I_D is traversed first in a topological sort of the graph, then the extensions E_i are compatible with each other. Further,
    1. If the input state S is compatible with the empty world, i.e., $S.Def = \emptyset$, then the extensions are compatible with the base document D, and may be loaded in the topologically-sorted order.
    2. Otherwise, the extensions are incompatible with the base document, and some required resources are missing, which are then reported.
5. If the graph is acyclic, but I_D is *not* traversed first, then somehow an extension defines something that D relies upon. This cannot happen with the current overlay language (but see Section 6 where it may occur).
6. Else the graph is cyclic, and so report the extensions contributing to a cycle as conflicting.

This algorithm is only as useful as its input is precise: it assumes that all semantic constraints have been encoded appropriately as guards on the relevant overlays. If any constraints are not so encoded, some topologically-sorted orderings may actually be "bad" while others are "good", though nothing in the conflict graph distinguishes them. This may cause false negatives: for some incompatible extensions, the algorithm will claim the conflict graph is acyclic and hence that some extensions are compatible; properly modeling all semantic constraints would yield a cyclic conflict graph.

4.5 Heuristics to determine optional composition order

The conflict-graph algorithm above takes guarded overlays as input. These have the simple property that they either successfully apply or fail. However, extension authors define *compositions*, not merely guarded overlays, and compositions may contain optional

(or one-of-several) components that can "successfully apply" by silently doing nothing. As we've seen, optional and one-of-several components are useful for expressing higher-level structural design constraints on the guarded overlays within an extension. Supporting such constructions, however, complicates the definition of when two guarded overlays conflict: for instance, an optional component can *always succeed* by doing nothing!

Extension authors would not bother to write an optional component and expect it always to do nothing; the intent is for it to apply whenever possible, but not to fail if it cannot do so. The goal must therefore be to compute a maximal set of optional components from source extensions that, when combined with the extensions' non-optional components and when treated as non-optional themselves, succeed in finding a compatible loading order. Unfortunately, this task is substantially harder than before: with n optional components, there are 2^n subsets to try, to see whether they can be loaded compatibly.

In fact, it is straightforward to prove that selecting such a subset is at least NP-hard:

Theorem 1. *Given a set of compositions $\{c_1, \ldots, c_n\}$ that may contain optional (?) or one-of-several (!) clauses, determining whether there exists a compatible loading order $c_{\pi(1)}, \ldots, c_{\pi(n)}$ (for some permutation π) is NP-hard.*

Proof sketch. Encode the clauses of a 3-CNF-SAT instance as a set of optional compositions and guarded overlays, and the formula itself as one additional, non-optional composition. It will successfully apply only if some subset of the optional compositions can be loaded compatibly; this subset induces a solution to the original 3-CNF-SAT problem. □

Since an exact solution is in general infeasible, any effective heuristic algorithm is appropriate instead. (This arises in practice: non-negligible numbers of users install over ten extensions [13]; exhaustively testing a worst case of over 2^{10} subsets is too slow.) We use a greedy heuristic, starting with all optional components and removing one arbitrary, conflicted component at a time until a compatible loading order is found. If we remove all optional components and still have not found a valid loading order, we report an error.

5. Firefox case-study

We applied the analysis defined in the previous section to a corpus of 350 Firefox extensions (the top-ranked extensions as of November 2008), all of which claimed compatibility with Firefox 3.0 and so could conceivably be installed simultaneously. Additionally, many extensions claimed compatibility with other Mozilla products, and so included overlays that were not intended for Firefox. (Precise details of the case study can be found in the first author's thesis [9, appendix B].)

As noted earlier, our analysis must support several idiosyncrasies of XUL and Firefox's implementation to model Firefox's behavior faithfully. Not handling these quirks leads to unacceptably many false positives in the analysis: too many extensions were claimed to conflict when in fact they were compatible. Note that there may be false negatives in the analysis (i.e., extensions are claimed to be compatible when in fact they conflict) only if it neglects to model some semantic constraint on resources. For example, the algorithm guarantees no false negatives with regard to *Key* and *Id* constraints.

5.1 Handling XUL idiosyncrasies

The intuitive understanding of XUL overlays is easily representable in the overlay language. A concrete XUL overlay

⟨**overlay**⟩
⟨**tag1** id="id1" *attrs*⟩*content1*⟨/**tag1**⟩
more actions
⟨/**overlay**⟩

might be modeled by the abstract overlay

Overlay(*Insert*(`tag1#id1`, end, *content1*),
Modify(`tag1#id1`, *attrs*),
more actions)

which highlights the dual actions of modifying the node's attributes and inserting new content into its subtree. Unfortunately, the actual implementation of XUL overlays does not match this simple intuition, in two ways.

Recursive overlays: First, Gecko appears to treat *all descendants* of the ⟨**overlay**/⟩ as potential actions, rather than merely its children. This makes it syntactically impossible to distinguish new nodes being contributed by the overlay from potentially-existing nodes that are intended targets. For example, the SpeedDial extension contains the following excerpt, which deals with new popup menu items:

⟨**popupset** id="mainPopupSet"⟩
⟨**popup** id="contentAreaContextMenu"⟩
...actual overlay code...
⟨/**popup**⟩
⟨**popup** id="speedDialButtonMenu"...⟩
...new code...
⟨/**popup**⟩
⟨/**popupset**⟩

Firefox defines both the ⟨**popupset** id="mainPopupSet"/⟩ element and its child ⟨**popup** id="contentAreaContextMenu"/⟩: the intent of this overlay is clearly to overlay that popup with additional menu items. However, the following ⟨**popup** id="speedDialButtonMenu"/⟩ is a *new* popup menu, and is intended to overlay the ⟨**popupset** id="mainPopupSet"/⟩. These two ⟨**popup**/⟩ nodes impose different constraints on the overlay, yet are syntactically indistinguishable. The "corrected" overlays to achieve these two behaviors should be

⟨**popup** id="contentAreaContextMenu"⟩
...actual overlay code...
⟨/**popup**⟩
⟨**popupset** id="mainPopupSet"⟩
⟨**popup** id="speedDialButtonMenu"...⟩
...new code...
⟨/**popup**⟩
⟨/**popupset**⟩

where now the named children of the target node must never previously exist. In our dataset, this pattern occurs over sixty times, and must be manually removed prior to analysis.

Higher-order behavior: Second, the overlay-loading process permits "higher-order overlays", where nodes in an overlay target other nodes in that same overlay, or in another overlay document, rather than the woven base document, two emergent properties of overlays that are not implied by the documentation for overlays [11]. Some extensions take advantage of this behavior: for example, Stylish uses the following snippet

⟨**keyset** id="mainKeyset"⟩
⟨**key** id="stylish-open-manage"/⟩
⟨/**keyset**⟩
⟨**key** id="stylish-open-manage" *more attrs...*/⟩

to produce

⟨**keyset** id="mainKeyset"⟩
⟨**key** id="stylish-open-manage" *more attrs...*/⟩
⟨/**keyset**⟩

as opposed to two separate key handlers, one of which fails to define a key! A straightforward translation of the former overlay into the abstract language yields

Overlay(*Insert*(`keyset#mainKeyset`, end,
⟨**key** id="stylish-open-manage"/⟩),
Modify(`keyset#mainKeyset`),
Insert(`key#stylish-open-manage`, end),
Modify(`key#stylish-open-manage`,
more attrs))

which has the wrong interface: the latter two actions require `key#stylish-open-manage` to be defined, but it is defined within this *Overlay* itself. Rather than contort our analysis, we manually rewrote overlays to remove this idiom [9].

5.2 Results and performance

Detected conflicts: Our analysis proceeds by processing each Firefox source file independently, examining all the overlays that are declared to target it. For example, of the 350 extensions we examined, 261 of them declare 331 overlays targeting `chrome://browser/content/browser.xul`, the main Firefox browser window. We analyze the target files independently because Firefox gives no guarantee that extensions are loaded atomically; instead, overlays are applied on demand, when the target files are parsed and displayed to the user. Thus the load order for `browser.xul` may be different than for `preferences.xul`.

The 350 extensions attempt to install a total of 1121 overlay files, of which 18 overlays purport to overlay a non-existent target file; several of these target other Mozilla applications, and the rest truly are erroneous. Note that no individual Mozilla application can detect these errors, because no Mozilla application is bundled with information about the structure of all the other Mozilla applications.

Once such mistaken overlay declarations and other quirks were accommodated, we found a dozen problematic extensions. Four points jump out from these results. First, the algorithm correctly detects three pairs of differently-versioned extensions (Sage and Sage Too, CaptureIt 1.0 and 2.5, and ForecastFox 0.9.7.7 and ForecastFox˙l10n 0.7) as conflicting with each other. This is a valuable sanity check that the algorithm is properly computing intersections.

Second, there are only twelve conflicting pairs of extensions, out of nearly 35,000 possible conflicting pairs. This may reveal a heavy selection bias in the sample: we examined the *top 350 most popular* extensions. (Alternate samplings would either deliberately seek out buggy extensions, inflating the reported conflict rate, or sample randomly and thereby deflate the reported rate.) Regardless of whether their conflict-freedom led to their popularity or vice versa, popular extensions work surprisingly well with others.

Third, the algorithm can detect simple typos that would otherwise be silently ignored by Firefox, or targets that are no longer defined. These are ignored by design, because they *might* be correct for other Mozilla applications or other versions, but this design choice prevents extension authors from detecting simple but subtle bugs in their code. To be sure, warnings about typos could likely be detected by a much simpler system; the analysis presented here incorporates them while providing additional benefit.

Finally, and most importantly, the effort needed to model the effects of previous extensions is warranted, because extensions do conflict with and extend each other in reality. Without this precision, the algorithm would miss true conflicts due to cycles of length greater than two, such as between MiniMap Sidebar, Toolbar Buttons, and Firefox itself. Likewise, it would produce false negatives and complain that an extension such as FireCookie (which depends on Firebug) always had missing dependencies, even if they were present.

The potential for false positives is highlighted by a trio of non-conflicting extensions. For `chrome://firebug/content/firebugOverlay.xul` (the main Firebug panel UI), both YSlow and FireCookie extend Firebug's main toolbar with an additional item. However, their overlays only work thanks to the quirks of Firefox's overlay loader mentioned above: to an analysis that strictly enforces XUL overlay semantics, it appears that both YSlow and FireCookie actually define a node (with id `"fbToolbarInner"`) that Firebug itself defines, when in fact Firefox treats that node as an overlay target. Further, because of how Firebug factored its code, it also appears that `"fbToolbarInner"` is not even defined! Without a sufficiently precise accounting of all three extensions, and a sufficient encoding of Firefox's quirks, the analysis might detect two distinct problems, neither of which in fact exist.

Performance: The conflict analysis is inherently cubic: it must compare the actions in the interfaces of every pair of extensions to compute potential conflict-graph edges. We ran the algorithm on all 264 overlays applicable to the main Firefox UI file at once; this is by far the largest target document and an order of magnitude more overlays than typical users will have installed. Nevertheless, the analysis completed in under four minutes, with 1.2GB of peak memory usage. This is sufficiently quick for extension developers to use regularly to ensure their work is compatible with most extensions.

The common case for end-users, however, is much faster. More typical end-user workloads of up to 25 extensions [13] ($10\times$ smaller than our test) will see roughly a $1000\times$ improvement, for a runtime of a few seconds and memory usage of a few dozen megabytes. Additionally, the implementation has not been highly optimized; many expensive set-intersection tests can be memoized or optimized away. Further, the results of the compatibility analysis need not be repeated unless the set of installed extensions changes; they can be cached easily and compactly the rest of the time. Finally, these compatibility checks can be cached by Mozilla for all users, amortizing the cost to practically nothing.

6. Enhancements to overlays, and challenges to analysis

Sections 3 to 5 presented the challenges facing Firefox-like overlays. But the abstract overlay language of Fig. 2 can generalize in several ways. We focus on just one here: using full *CSS selectors* to choose target nodes. This greatly improves the expressiveness of overlays, but also greatly complicates the conflict-detection algorithm. As no existing system implements this generalization yet, the results in this section are a warning of the pitfalls ahead.

Currently, overlay actions can target precisely one node at a time: that unique node with the given name and id. However, extensions may reasonably want to target multiple nodes in a uniform way; we motivate this flexibility with two examples, one simplified from real-world extension idioms and one that highlights why such flexibility is distinctly different from the versions of this language examined above.

The ability for selectors to match multiple nodes introduces a new and difficult problem: nodes can now match several different selectors, rather than just the sole selector determined by the tagname and id of the node. Conversely, it is possible for two distinct selectors to *intersect*, such that there exist nodes matching them both. It is therefore now possible for two overlays with different selectors to in fact apply to the same targets, and hence potentially conflict.

Checking for the overlap of two selectors can be done efficiently, and using it, the conflict-graph analysis defined above can be adapted to provide *partial* support for these new selectors. The use of the intersection algorithm complicates the maintenance of

the *Undef* sets of the analysis; additionally, the CSS universal selector must be handled differently than others. We explain the adaptation of the analysis, and then explain these two caveats. Fully adapting the analysis to remove these problems remains as future work.

6.1 Motivating examples

Refactoring a single extension: Consider an extension (simplified from real examples) that adds a submenu of actions to the "Tools" menu of some application, and adds an identical submenu to the application's context menu. One way to write the extension might be to duplicate the submenu's contents in two overlays (written here using XUL):

OV12*: *Overlay*(*Insert*(`menu#tools-menu`, end,
⟨**submenu** id="aMenu"⟩
⟨**menuitem**⟩Hi⟨/**menuitem**⟩
⟨/**submenu**⟩))

OV13*: *Overlay*(*Insert*(`menu#context-menu`, end,
⟨**submenu** id="aMenu"⟩
⟨**menuitem**⟩Hi⟨/**menuitem**⟩
⟨/**submenu**⟩))

However, such copy-and-paste duplication easily leads to divergences between the two versions. An alternate approach might permit extensions to specify a set of selectors:

Overlay(*Insert*({`menu#tools-menu`,
`menu#context-menu`}, ...))

Such an overlay would apply to all nodes matching any element in that set. However, this is a fairly limited improvement, as the extension author must enumerate all targets explicitly, which may not always be possible, as the next example will show. A better approach (and, indeed, one taken by some Firefox extensions), is to refactor the common code (in this case, the ⟨**menuitem**/⟩) into a *third* overlay:

OV12**: *Overlay*(*Insert*(`menu#tools-menu`, end,
⟨**submenu** id="myMenu"/⟩))

OV13**: *Overlay*(*Insert*(`menu#context-menu`, end,
⟨**submenu** id="myMenu"/⟩))

OV14*: *Overlay*(*Insert*(`submenu#myMenu`, end,
⟨**menuitem**⟩Hi⟨/**menuitem**⟩))

Here, overlays OV12** and OV13** create *stubs* that can be filled in by OV14*. As written, however, these overlays violate the well-formedness property of the combined document, since two ⟨**submenu**/⟩ items with the same id will be created. The best approach, then, is to use a property other than ids for OV14*:

OV12: *Overlay*(*Insert*(`menu#tools-menu`, end,
⟨**submenu** class="myMenu"/⟩))

OV13: *Overlay*(*Insert*(`menu#context-menu`, end,
⟨**submenu** class="myMenu"/⟩))

OV14: *Overlay*(*Insert*(`submenu.myMenu`, end,
⟨**menuitem**⟩Hello⟨/**menuitem**⟩))

Such an approach requires more flexible selectors than we've permitted so far. This seemingly-small improvement—after all, this is still merely a simple selector—provides a large expressive jump, as such selectors can match *multiple targets*.

Supporting relationships between nodes: Suppose an extension wanted to render lists of items as set notation, by surrounding them with braces and inserting commas between the items. Such an overlay might be written using

Overlay(*Insert*(`ul`, start, ⟨**span**⟩{⟨/**span**⟩),
Insert(`ul > li ~ li`, before, ⟨**span**⟩, ⟨/**span**⟩),
Insert(`ul`, end, ⟨**span**⟩}⟨/**span**⟩)))

The first and last actions insert the list braces, while the middle action inserts commas only before those list items with at least one preceding sibling, i.e., every list item except the first.[3] This overlay uses a non-simple selector (the child and sibling combinators), and so marks a large expressive jump from the languages in the previous sections.

6.2 CSS selector intersection

Generalizing to arbitrary CSS selectors means not only that selectors can match multiple nodes, but that nodes can be matched by several distinct selectors. Therefore to detect when two overlays might interfere, we need to know whether, for a given pair of selectors, there could exist a tree with elements matching both selectors simultaneously. Such language intersection problems are common (cf. regular expression intersections [1, 4], XQuery and XPath intersection [2, 3, 6], etc.), and for arbitrary (context-free or larger) languages these problems are undecidable [12]. Fortunately, in prior work we showed that CSS is at heart a regular language, and so the intersection problem is decidable.

The details of constructing the precise intersection of two selectors are intricate: in general, the intersection cannot be described by a single selector, but rather only by a set of them. While the full algorithm is exponentially expensive, we can test it for (non-)emptiness very efficiently, which is all that is needed for the conflict-detection algorithm [9].

6.3 Impact on the conflict-detection algorithm

The conflict-graph construction algorithm must be adapted to handle the enhanced expressive power of these selectors: it is no longer enough simply to check for set intersections. Consider a base document consisting solely of the empty node ⟨**a**/⟩, and the following three overlays

OV15: *Overlay*(*Insert*(a, end, ⟨**b** class="x y"/⟩),
Insert(a, end, ⟨**c** class="y"id="idC"/⟩),
Forbid(*Sel*(`d.foo`)))

OV16: *Overlay*(*Insert*(`b.x`, end, ⟨**d** class="foo"/⟩))

OV17: *Overlay*(*Insert*(`*.foo`, end, ⟨**e**/⟩))

The compositions OV15 ; OV16 and OV15 ; OV17 are both feasible and demonstrate two challenges in dealing with general selectors, both due to the possibility of *aliasing*.

Handling Undef sets A careful translation of OV15's and OV16's interfaces yields the following sets:

$$
\begin{aligned}
S_i^{15}.Def &= \{Sel(\texttt{a})\} \\
S_i^{15}.Undef &= \{Id(\texttt{idC}), Sel(\texttt{d.foo})\} \\
S_o^{15}.Def &= \{Sel(\texttt{a}), Sel(\texttt{a > b.x.y}), Id(\texttt{idC}), \\
&\qquad Sel(\texttt{a > b.x.y + c.y\#idC})\} \\
S_o^{15}.Undef &= \{Sel(\texttt{d.foo})\} \\
S_i^{16}.Def &= \{Sel(\texttt{b.x})\} \\
S_o^{16}.Def &= \{Sel(\texttt{b.x}), Sel(\texttt{b.x > d.foo})\}
\end{aligned}
$$

These sets satisfy Eqs. (1) to (4), so the current algorithm would decide OV15 ; OV16 is feasible, and compute S_i and S_o for the

[3] The HTML 5-expert reader will notice that these three insertions produce a technically malformed ⟨**ul**/⟩, but this is no worse than what scripts can already create at runtime. The CSS 3-expert reader will notice that these insertions can be mimicked by generated content, but such content is not selectable and does not behave as regular text.

composition. But doing so yields

$$\begin{aligned} S_i.Def &= S_i^{15}.Def \cup (S_i^{16}.Def \setminus \text{defs}(\text{OV15})) \\ &= \{Sel(\texttt{a}), Sel(\texttt{b.x})\} \end{aligned}$$

This implies that OV15 ; OV16 can apply only to a document already containing a node matching `b.x`—but OV15 defines such a node! This extra constraint may falsely prevent OV15 ; OV16 from applying. The problem arises because the set-difference operation is too restrictive: it compares elements by equality, rather than by overlap. OV15 defines `a>b.x.y`, which intersects with—and therefore satisfies—the `b.x` requirement of OV16.

Similarly, the set-intersections in Eqs. (1) to (4) are likewise too restrictive. The algorithm *should* decide the composition order OV16 ; OV15 is infeasible, because OV15 forbids documents containing nodes matching `d.foo`. But the two relevant selectors, $Sel(\texttt{d.foo}) \in S_i^{15}.Undef$ and $Sel(\texttt{b.x>d.foo}) \in S_o^{16}.Def$ are not equal, though they do intersect; a set intersection would unhelpfully decide these two sets are disjoint and incorrectly accept this composition.

As a result, sequencing must be revised to use pairwise tests for non-intersection: for two overlays OV1 and OV2, and their interfaces S^1 and S^2, Eqs. (1) to (4) become

$$\forall d_2 \in S_o^2.Def, \forall u_1 \in S_i^1.Undef, d_2 \between u_1 = \emptyset \quad (5)$$

$$\forall d_2 \in S_o^2.Def, \forall d_1 \in \text{defs}(\text{OV1}), d_2 \between d_1 = \emptyset \quad (6)$$

$$\forall f_2 \in S_o^2.Frozen, \forall u_1 \in \text{used}(\text{OV1}), f_2 \between u_1 = \emptyset \quad (7)$$

$$\forall r_2 \in \text{reqs}(\text{OV2}), \forall c_1 \in S_i^1.Clean, r_2 \between c_1 = \emptyset \quad (8)$$

(Here, $s \between t$ denotes the intersection of two selectors.) These equations strictly generalize Eqs. (1) to (4). But similarly changing the set-difference operation introduces false positives: many CSS selectors now may produce unexpectedly non-empty intersections. Finding an appropriate analogue for set-difference with general selectors is open work.

Handling universal selectors: A similar problem arises with universal selectors. For example, overlay OV17 applies to nodes matching `*.foo`. Therefore $Sel(\texttt{*.foo})$ must be included in both $S_i^{17}.Def$ and $S_o^{17}.Def$. But this exposes an ambiguity in the meaning of the universal selector. In the input set, `*.foo` means "any node with class `foo`," while in the output set it means "*some* node with class `foo`." But for the intersection algorithm, it takes only the former meaning. In particular, suppose the analysis tried to compose OV15 ; OV17. Then since $S_o^{15}.Undef = \{Sel(\texttt{d.foo})\}$ and $S_i^{17}.Def = \{Sel(\texttt{*.foo})\}$, which clearly overlap, Eq. (5) would erroneously conclude that OV17 cannot follow OV15. In fact, so long as some element existed with class `foo` and tagname not `d`, both OV15 and OV17 would be satisfied.

Properly handling these two meanings of `*` requires going outside the CSS selector language, and is also future work.

7. Related work

While overlays resemble aspects in several ways, their conflict-resolution algorithms look very different. Here, we focus instead on two related areas of research that contend with overlays' key actions of describing regions in a document and describing changes to them.

Context logic and dynamic overlays Context logic [5, 7] reasons about mutations of tree-shaped data. It employs a "small-footprint" approach similar to the overlay interfaces above, where modifications mention only the part of the structure that they touch. The structure of the logic then explores how the context of one modification is changed by the effects of another, earlier modification, which may resolve the ambiguities described above for overlay interfaces. For example, if one overlay ensures that the document matches `a+b` and a second overlay inserts $\langle \mathbf{c}/\rangle$ after the $\langle \mathbf{a}/\rangle$, the composite document no longer satisfies `a+b`, but rather `a+c+b`. It is unclear how to encode CSS selectors into context logic formulas such that the context will transform appropriately.

Patch theory Patch theory [8, 10] defines an algebra of patches, and was invented to give a formal foundation to text-based patch tools. The algebra describes how patches commute with one another, and requires the existence of an inverse for every possible patch action. As presented, overlays have no inverses, but if we permit them to *remove* nodes as well as add new ones, such inverses do exist. As yet, no theory of patches on tree-shaped data exists.

References

[1] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51 – 72, 1995.

[2] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys (CSUR)*, 41:3:1–3:54, Jan. 2009.

[3] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)*, 55(2):1–79, 2008.

[4] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11:481–494, Oct. 1964.

[5] C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjunct elimination in context logic for trees. In Z. Shao, editor, *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 255–270. Springer Berlin / Heidelberg, 2007.

[6] J. Cheney. Satisfiability algorithms for conjunctive queries over trees. In *International Conference on Database Theory (ICDT)*, 2011.

[7] P. Gardner and U. Zarfaty. An introduction to context logic. In D. Leivant and R. de Queiroz, editors, *Logic, Language, Information and Computation*, volume 4576 of *Lecture Notes in Computer Science*, pages 189–202. Springer Berlin / Heidelberg, 2007.

[8] J. Jacobson. A formalization of darcs patch theory using inverse semigroups. Technical Report (09-83), UCLA Computational and Applied Mathematics, Oct. 2009. `http://www.math.ucla.edu/~jjacobson/patch-theory/`.

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

[10] I. Lynagh. Darcs patch theory (more or less). Originally posted to darcs-users mailing list, Sept. 2008. `http://lists.osuosl.org/pipermail/darcs-users/2008-August/013040.html`.

[11] Mozilla. XUL overlays. Written Jan. 2010. `https://developer.mozilla.org/en/XUL_Overlays`.

[12] M.-J. Nederhof and G. Satta. The language intersection problem for non-recursive context-free grammars. *Information and Computation*, 192(2):172 – 184, 2004.

[13] J. Scott. How many Firefox users have add-ons installed? 85%! Written June 2011. `http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/`.

A. Details of overlay interfaces

$$\mathrm{reqs}(\mathit{Overlay}(\vec{a})) \stackrel{\text{def}}{=} \bigcup\{\mathrm{reqs}(a_i) \mid a_i \in \vec{a}\}$$

$$\mathrm{reqs}(\mathit{Insert}(s, \_, \_)) \stackrel{\text{def}}{=} \{\mathit{Selector}(s)\}$$

$$\mathrm{reqs}(\mathit{Modify}(s, \_)) \stackrel{\text{def}}{=} \{\mathit{Selector}(s)\}$$

$$\mathrm{defs}(\mathit{Overlay}(\vec{a})) \stackrel{\text{def}}{=} \bigcup\{\mathrm{defs}(a_i) \mid a_i \in \vec{a}\}$$

$$\mathrm{defs}(\mathit{Insert}(s, \_, \vec{h})) \stackrel{\text{def}}{=} \bigcup\{\text{selectors describing each } h_i \in \vec{h}\}$$

$$\mathrm{defs}(\mathit{Modify}(\_, \_)) \stackrel{\text{def}}{=} \emptyset$$

$$\mathrm{used}(o) \stackrel{\text{def}}{=} \mathrm{reqs}(o) \cup \mathrm{defs}(o)$$

OVERLAY-INTERFACE

$$\frac{S_i = \left\{ \begin{array}{r} \mathit{Def} = \mathrm{reqs}(o) \\ \mathit{Undef} = \mathrm{defs}(o) \end{array} \right\}, S_o = \{\ \mathit{Def} = \mathrm{used}(o) \quad \}}{[\![S_i]\!]\ o\ [\![S_o]\!]}$$

G-REQUIRE

$$\frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_i' = S_i \cup \{\mathit{Def} = \vec{r}\} \quad S_o' = S_o \cup \{\mathit{Def} = \vec{r}\}}{[\![S_i']\!]\ \mathit{Require}(\vec{r}, g)\ [\![S_o']\!]}$$

G-FORBID

$$\frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_i' = S_i \cup \{\mathit{Undef} = \vec{r}\} \quad S_o' = S_o \cup \{\mathit{Undef} = \vec{r} \setminus S_o.\mathit{Def}\}}{[\![S_i']\!]\ \mathit{Forbid}(\vec{r}, g)\ [\![S_o']\!]}$$

G-FIRST

$$\frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_i' = S_i \cup \{\mathit{Clean} = \vec{r}\}}{[\![S_i']\!]\ \mathit{First}(\vec{r}, g)\ [\![S_o]\!]}$$

G-LAST

$$\frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_o' = S_o \cup \{\mathit{Frozen} = \vec{r}\}}{[\![S_i]\!]\ \mathit{Last}(\vec{r}, g)\ [\![S_o']\!]}$$

Figure 3: Defining the interface for a guarded overlay

S-SEQUENCE

$$\frac{\begin{array}{c} [\![S_i^1]\!]\ c_1\ [\![S_o^1]\!] \qquad [\![S_i^2]\!]\ c_2\ [\![S_o^2]\!] \\ S_o^1.\mathit{Def} \cap S_i^2.\mathit{Undef} = \emptyset \qquad S_o^1.\mathit{Def} \cap \mathrm{defs}(c_2) = \emptyset \\ S_o^1.\mathit{Frozen} \cap \mathrm{used}(c_2) = \emptyset \qquad S_i^2.\mathit{Clean} \cap \mathrm{reqs}(c_1) = \emptyset \\ S_i = \left\{ \begin{array}{r} \mathit{Def} = S_i^1.\mathit{Def} \cup (S_i^2.\mathit{Def} \setminus \mathrm{defs}(c_1)) \\ \mathit{Undef} = S_i^1.\mathit{Undef} \cup S_i^2.\mathit{Undef} \\ \mathit{Clean} = S_i^1.\mathit{Clean} \cup S_i^2.\mathit{Clean} \\ \mathit{Frozen} = \emptyset \end{array} \right\} \\ S_o = \left\{ \begin{array}{r} \mathit{Def} = S_o^1.\mathit{Def} \cup S_o^2.\mathit{Def} \\ \mathit{Undef} = S_o^2.\mathit{Undef} \cup (S_o^1.\mathit{Undef} \setminus \mathrm{defs}(c_2)) \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = S_o^1.\mathit{Frozen} \cup S_o^2.\mathit{Frozen} \end{array} \right\} \end{array}}{[\![S_i]\!]\ c_1\ ;\ c_2\ [\![S_o]\!]}$$

Figure 4: Semantics of sequencing