

# Local Reasoning about Mashups

Philippa Gardner, Gareth Smith, and Adam Wright

Imperial College London, UK  
{pg, gds, adw07}@doc.ic.ac.uk

**Abstract.** Web mashups are complex programs that dynamically compose data and code from many sources. Whereas data is sometimes formally specified by XML schema, code never is. This makes it difficult to construct reliable software. Using local Hoare reasoning, introduced in separation logic to reason about e.g. C programs and extended in context logic to reason about e.g. the DOM library, we are able to reason about mashup programs, proving that they are fault-free and providing specifications for code that are analogous to XML schema for data.

## 1 Introduction

The growing “mashup” phenomenon involves websites using JavaScript alongside XML to create complex web applications that integrate data and code from many sources. This leads to problems with reliability: sources may change unaware that remote code depends on them; clients may misuse a service due to lack of specification; or two sources may conflict due to incompatible use of resource.

Building on our work specifying the W3C DOM library for XML update, we show how to give specifications to sources. These specifications describe both the shape of the source’s data (analogous to XML schema) and the behaviour of the code it provides (a kind of schema for code). With such specifications, one can construct provably fault-free mashups, where services deliberately expose a subset of their resource, and clients ensure the integration of sources is sound.

We will illustrate our ideas using a mashup example based on the popular Twitter Service. Twitter is a service where users can share textual updates. Twitter exposes the update list as XML. Subsets of this data are often embedded within other websites: for example, our research group homepage might list all the recent Twitter updates people have targeted at us. Unfortunately, the update list can be polluted with spam, which could therefore appear on our homepage. One solution is to use a spam checker service to filter known spammers from the list before we display it. This combination of three elements (our homepage, the Twitter updates data, and the spam filtering system) forms a mashup (see figure 1).

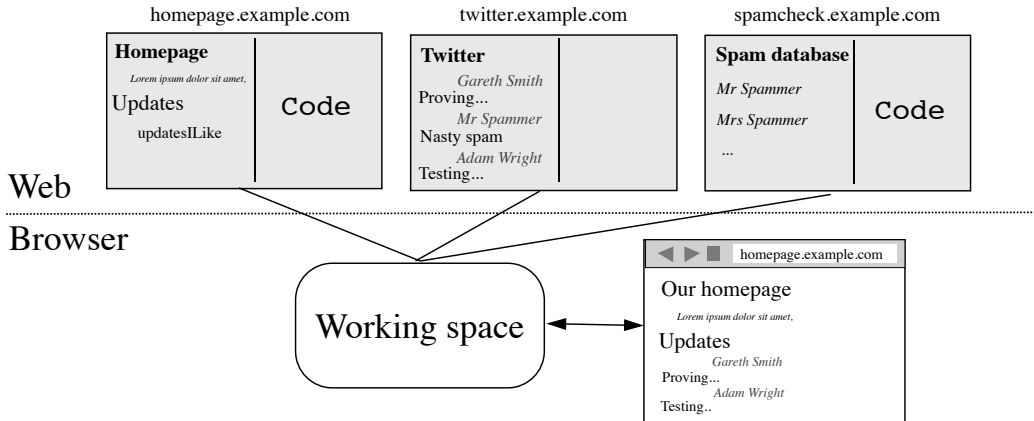


Fig. 1. Twitter mashup example

The figure shows the browser displaying homepage.example.com. The browser is directed by the code of homepage to get the data from homepage and Twitter, and data and code from the spam checker, and form

the mashup. In reality, the data and code will be web pages - a mixture of XML and JavaScript code, possibly spread over several files. Here, we work with data and code that has already been parsed by the browser into DOM data and separate code. We are able to reason about the homepage code, assuming published specifications for the data and code being pulled from the web. In our running example, we will prove that the homepage code is fault-free and will result in data fitting the schema of the example in figure 1, under the assumptions that the homepage and Twitter data match schemas for their examples, and that the spam checker code requires a string (a person's name) and returns a boolean. We require no additional assumptions about how the spam checker works.

Our reasoning is based on local Hoare reasoning, introduced in separation logic [1] to reason about e.g. C programs and extended in context logic to reason about e.g. the DOM library. The work on DOM mainly focused on the specification rather than exploring reasoning about programs using DOM. Here we apply the reasoning to mashup code, which makes extensive use of the DOM library.

Based on an analysis of web programs, we have identified central constructs of JavaScript necessary for mashup programs. We have combined our formal DOM specification with reasoning about these programming constructs. We are currently testing our language and reasoning with many examples arising from the web. We have noted that the mashup behaviour of these examples tends to be similar, and so here present our Twitter homepage as an illustration of this common pattern.

In this running example, we use assertions to specify the shape of the homepage and Twitter updates. We specify that the spam checker code requires a string (a person's name) and returns a boolean. We make no additional assumptions about how the spam checker works. With these specifications we prove that the homepage code is fault-free and that the only change to the homepage data is the addition of the Twitter updates.

**Related work:** There are several pieces of work which apply techniques from formal methods to DOM and JavaScript. The ideas in this paper build on Gardner and Smith's previous work on formally specifying DOM [2, 3], which followed Thiemann's work on a type safe DOM [4]. Our language is very much inspired by features from JavaScript. Maffeis, Mitchell and Taly have given an operational semantics of JavaScript [5], which significantly aided our understanding of JavaScript. There is various work on adding XML schema-like types to programming languages (e.g. XDuce [6]). We follow the view of Vitek [7], who argues that typing JavaScript is not a good idea, and we choose to work instead on the pre/post reasoning of Hoare.

There is a recent body of work which restricts the mashup environment to make it safe to put guest code on any web page: examples include Adsafe [8], Caja [9] and [10, 11]. As far as we are aware there is no published work on using logical specifications to tackle the faulting code and dependency problem for mashups. The nearest work to ours is an unpublished note of Klose and Ostermann on modular logic metaprogramming [12]. They develop a traditional module system, where the interfaces between modules are described as logical formulae, with the consistency and compatibility amongst modules also expressed in terms of logical consequence. Our work centres on the dynamic world of mashups. It will however be very interesting to assess the benefit of adapting our local reasoning approach to standard modules.

## 2 Reasoning about Featherweight DOM

We begin with a summary of the Featherweight DOM work of Gardner *et al* [2]. The DOM API is a W3C standardised library [13] for manipulating tree data and is ubiquitous in web programming: for example, all browsers provide an implementation for manipulating web page structure. Featherweight DOM provides an axiomatic semantics for a fragment of the DOM API (a fragment of the node interfaces which concentrates on the XML tree structure), together with local reasoning for programs using this simplified DOM API. The work has recently been extended to give a full axiomatic specification of the fundamental interfaces of DOM Core level 1 [3].

**Language:** The language of Featherweight DOM in figure 2 is a simple imperative WHILE language, augmented with a selection of commands from the W3C DOM standard. Analogous to the languages used in separation logic papers, we have both a variable store and a heap. The store is standard. Rather than the flat heap model of separation logic, our heap model is essentially the DOM data model. The DOM commands manipulate this tree-shaped heap.

<pre> Statement ::=   Id = IdExpr   Str = StrExpr   Int = IntExpr   Bool = BoolExpr   if ( BoolExpr ) then { Statement* }   else { Statement* }   while ( BoolExpr ) { Statement* }   skip   Command   Statement; Statement </pre>	<pre> Command ::=   Id = createElement(IdExpr, StrExpr)   Str = getNodeName(IdExpr)   Id = getParentNode(IdExpr)   Id = getChildNodes(IdExpr)   Id = appendChild(IdExpr, IdExpr)   Id = removeChild(IdExpr, IdExpr)   Id = item(IdExpr, IntExpr)   ... </pre>
--	---

Fig. 2. Featherweight DOM Grammar

With this language, we can write programs that construct and analyse DOM trees using the DOM API: for example,

```

updates = getChildNodes(node);
thirdkid = item(updates, 2);
appendChild(someUpdates, thirdkid)

```

This program obtains the list of children for the tree node identified by `node`. It extracts the third child in this list, storing the identifier in `thirdkid`. It then moves the node identified by `thirdkid` to be the last child of the node identified by `someUpdates`. This program will only run correctly on certain DOM trees. Such trees must contain a node identified by `node` which has at least three children. Moreover, the third child must not be an ancestor of the node identified by `someUpdates`.

**Data structure:** The data structure defining our tree-shaped heap is given by a grammar defining tree nodes, forests (lists of children of a tree node), groves (heaps of routed trees) and strings.

$$\begin{aligned}
\text{Trees } \mathbf{t} &::= \mathbf{s}_{id}[\mathbf{f}]_{fid} \mid \#text_{id}[\mathbf{s}] \\
\text{Forests } \mathbf{f} &::= \emptyset_F \mid \langle \mathbf{t} \rangle_F \mid \mathbf{f} \otimes \mathbf{f} \\
\text{Groves } \mathbf{g} &::= \emptyset_G \mid \langle \mathbf{t} \rangle_G \mid \mathbf{g} \oplus \mathbf{g} \\
\text{Strings } \mathbf{s} &::= \emptyset_S \mid \mathbf{c} \mid \mathbf{s} \cdot \mathbf{s}
\end{aligned}$$

Notice that, as well as a tree node having an associated node identifier (`id`), its child list has a forest identifier (`fid`). These identifiers are analogous to heap addresses and must be unique across the entire data structure. This allows commands to address nodes directly, regardless of the current tree structure. We also work with single-holed contexts for all these structures, defined as standard with the holes annotated with the types of the removed data.

We define an equivalence on this tree-structured data:  $\otimes$  operator is associative with identity  $\emptyset_F$ ;  $\oplus$  is associative and commutative with identity  $\emptyset_G$ ; and  $\cdot$  is associative with identity  $\emptyset_S$ .

With this structure, we can give an represent a simple version of the Twitter update list. Written in XML, it has the form

```

<twitter>
  <update><user>Adam Wright</user> Hi</update>
  <update><user>Gareth Smith</user> Hello</update>
</twitter>

```

In this paper, we do not work with XML directly but rather its “parsed” form using a unspecified XML parser: for example, the above XML Twitter example has the parsed form:

$$\left\langle \begin{array}{l} \text{twitter}_{id1} [ \langle \text{update}_{id2} [ \langle \text{user}_{id3} [ \#text_{id4} [\text{Adam Wright}]_{id6} \otimes \#text_{id5} [\text{Hi}]_F ]_{id7} \otimes \\ \text{update}_{id8} [ \langle \text{user}_{id9} [ \#text_{id10} [\text{Gareth Smith}]_{id12} \otimes \#text_{id11} [\text{Hello}]_F ]_{id13} \rangle \\ \rangle ]_{id14} \end{array} \right\rangle_G \quad (1)$$

using a shorthand for string notation ( $\mathbf{F} \cdot \mathbf{o} \cdot \mathbf{o}$  is rendered as `Foo`). For brevity, we will often omit bracketing when it can be deduced; identifiers which one can assume exist and are unique; strings within text nodes; and the children of a node, which means an empty child list.

**Assertion language:** We define an assertion language to specify properties of our tree-structured heaps. Recall that assertion languages based on separation logic consist of three parts: classical first-order logic assertions, separating assertions constructed from the commutative separating conjunction  $*$  and its right adjoint  $\multimap$ , and

heap-specific assertions such as  $0$  denoting the empty heap, and  $x \mapsto y$  denoting a heap with exactly one cell. Here, we will use an assertion language based on context logic, which has a similar structure. In our case, the separating assertions are constructed from a non-commutative separating application  $\circ_D$ , and its two separating right adjoints  $\dashv$  and  $\circ_{-D}$ : the assertion  $P \circ_D Q$  specifies that a given tree can be split into two disjoint parts, a subtree of type  $D$  satisfying  $Q$  and a tree context satisfying  $P$ ; the assertion  $P \dashv Q$  specifies that, whenever a subtree satisfying  $P$  is put inside a given context, the result satisfies  $Q$ ; and the assertion  $P \circ_{-D} Q$  specifies that, whenever a context satisfying  $P$  is placed around the given tree, then the result satisfies  $Q$ .

As our “heap” is given by tree shaped data structure, we give assertions that describe these trees. Our tree assertions are defined by a lift of the data structure. For example, we can use these assertions to define an *update* predicate that accepts only update nodes with a user node and #text node as children:

$$\text{update} \triangleq \exists i_1, \dots, i_6, s_1, s_2. \text{update}_{i_1}[\langle \text{user}_{i_2}[\langle \# \text{text}_{i_3}[s_1] \rangle]_{i_4} \otimes \# \text{text}_{i_5}[s_2] \rangle]_{i_6}$$

Again, from now we may omit angle brackets when the meaning is clear, identifiers and strings when they have been existentially quantified, and the child list of nodes when it is empty. We also use  $\text{name}_{i_d}$ , to mean either a tree node with arbitrary children, or a text node.

We have the following logical equivalence between assertions, demonstrating the use of the separating application:

$$\text{update} \iff \text{update}[\dashv_{\text{T}} \otimes \# \text{text}] \circ_{\text{T}} \text{user}[\# \text{text}] \quad (2)$$

Using separating application, we can also derive the assertions  $\Box_{\otimes} P$  and  $\Diamond P$ : assertion  $\Box_{\otimes} P$  specifies a forest in which every tree satisfies  $P$ ; assertion  $\Diamond P$  specifies a forest with a subtree satisfying  $P$ . For example, we can compactly describe an arbitrary update list in Twitter, which requires that all the child trees of the twitter node are updates:

$$\text{twitter} \triangleq \text{twitter}[\Box_{\otimes}(\langle \text{true}_{\text{T}} \rangle_{\text{F}} \implies \langle \text{update} \rangle_{\text{F}})] \quad (3)$$

**Program reasoning:** Using our assertion language, we build a local Hoare Logic for reasoning about our DOM programs, in the style of separation logic. We use the O’Hearn “fault avoiding” interpretation of Hoare triples:  $\{P\}\mathbb{C}\{Q\}$  is semantically valid if in all states satisfying  $P$ ,  $\mathbb{C}$  cannot fault and when  $\mathbb{C}$  terminates, the program state is described by  $Q$ .

We combine standard Hoare rules with axioms and an abstract frame rule. The `getChildNodes` command has the axiom

$$\frac{\{ \text{name}_{\text{node}}[F]_{fid} \wedge kids = Y \} \quad \text{id} = \text{getChildNodes}(\text{node})}{\{ \text{name}_{\text{node}[Y/kids]}[F]_{fid} \wedge kids = fid \}}$$

This axiom is small because it specifies the behavior of `getChildNodes` just on its footprint, in this case the subtree identified by `node`.

The abstract frame rule is analogous to the separation frame rule, using separation application instead of separating conjunction. Let  $\mathbb{C}$  be the code `id = getChildNodes(node)`. Using the rule of consequence, abstract frame and an instance of the small axiom above (node is now just a program variable, not an arbitrary expression), we obtain the derivation for  $\mathbb{C}$ :

$$\frac{\frac{\{ \text{user}_{\text{node}}[F]_{fid} \} \mathbb{C} \{ \text{user}_{\text{node}}[F]_{fid} \wedge \text{id} = fid \}}{\text{AXIOM}} \quad \frac{\{ \text{update}[\dashv_{\text{T}} \otimes \# \text{text}] \circ \text{user}_{\text{node}}[F]_{fid} \wedge \text{id} = fid \} \mathbb{C} \{ \text{update}[\dashv_{\text{T}} \otimes \# \text{text}] \circ \text{user}_{\text{node}}[F]_{fid} \}}{\text{FRAME}}}{\{ \text{update}[\text{user}_{\text{node}}[F]_{fid} \otimes \# \text{text}] \} \mathbb{C} \{ \text{update}[\text{user}_{\text{node}}[F]_{fid} \otimes \# \text{text}] \wedge \text{id} = fid \}} \text{CONSEQUENCE}}$$

The use of abstract frame declares that the data described by the context remains unchanged.

The specification `appendChild(parent, newChild)` not only requires that `parent` and `newChild` are in the DOM tree, but requires that `newChild` is not an ancestor node of `parent`:

$$\frac{\{ (\emptyset_{\text{F}} \dashv (\text{cg} \circ_{\text{T}} (\text{name}_{\text{parent}}[\mathbf{f}]))) \circ_{\text{F}} \langle \text{name}'_{\text{newChild}} \wedge \mathbf{t} \rangle_{\text{F}} \} \quad \text{appendChild}(\text{parent}, \text{newChild})}{\{ \text{cg} \circ_{\text{T}} (\text{name}_{\text{parent}}[\mathbf{f}] \otimes \langle \text{name}'_{\text{newChild}} \wedge \mathbf{t} \rangle_{\text{F}}) \}}$$

The pre-condition describes data that can be separated into a context and the `newChild` node. The context is such that, if we were to place an empty forest in the hole made by removing the subtree rooted at `newChild`, we would still be able to extract the `parent`. This ensures that `parent` cannot be a descendant of `newChild` (if it was, there would be no way to extract the `parent` from the context left by removing `newChild` and its children). This specification shows the necessity of our contextual reasoning when specifying DOM.

Using our reasoning, we can specify the program given above. As well as showing the resource footprint the program uses, this proof also shows that when provided with the specified resource, it cannot fault.

$$\begin{aligned} & \{\exists s.\text{updates}_{\text{node}}[F_1 \otimes s[F_3] \otimes F_2 \wedge \text{len}(F_1) = 2] \oplus \text{others}_{\text{someUpdates}}[\emptyset_F]\} \\ & \quad \text{updates} = \text{getChildNodes}(\text{node}); \\ & \quad \text{thirdkid} = \text{item}(\text{updates}, 2); \\ & \quad \text{appendChild}(\text{someUpdates}, \text{thirdkid}) \\ & \{\exists s.\text{updates}_{\text{node}}[F_1 \otimes F_2 \wedge \text{len}(F_1) = 2]_{\text{updates}} \oplus \text{others}_{\text{someUpdates}}[s_{\text{thirdkid}}[F_3]]\} \end{aligned}$$

### 3 Reasoning about Mashups

The work on Featherweight DOM focusses on a formal axiomatic specification. It hardly explores reasoning about programs that call the DOM library. Here, we apply our reasoning to mashup programs, expanding our language, data structure and reasoning.

**Language:** Recall figure 1 and the five elements that are combined to form our homepage mashup. There are three distinct sources: data and code from `homepage.example.com`, data and code from `spamcheck.example.com` and data from `twitter.example.com`. Recall that, as we work with XML parsed into our data structure, we simplify this “page” notion by assuming that some other process has parsed the page to separate data and code. We refer to this post-parsing result as a mashup component.

A *mashup component* is a pair  $(\mathbb{D}, \mathbb{C})$ , where  $\mathbb{D}$  is grove data (a top-level tree or empty data) and  $\mathbb{C}$  is program code in our language. A *mashup*  $\mathbf{m}$  is a finite partial function mapping URIs to mashup components. Where before our language operated on the pair of a store  $\mathbf{s}$  and a heap  $\mathbf{h}$ , we now extend this pair with a mashup  $\mathbf{m}$ . We then extend our language with two new *mashup IO* commands:

`Id = fetchDocument(StrExpr)`: evaluates `StrExpr` to obtain a *uri*. If  $\mathbf{m}(\text{uri}) = (\mathbb{D}, \mathbb{C})$ , it takes  $\mathbb{D}$ , freshens the identifiers with respect to the current heap, appends the result to the heap at grove level, and stores the new root node identifier in `Id`. It faults if  $\text{uri} \notin \text{dom}(\mathbf{m})$ .

`runScript(StrExpr)`: evaluates `StrExpr` as above. If  $\mathbf{m}(\text{uri}) = (\mathbb{D}, \mathbb{C})$ , the command behaves as  $\mathbb{C}$ . It faults if  $\text{uri} \notin \text{dom}(\mathbf{m})$ .

Using these commands, we can begin to write the code of `homepage.example.com`. Consider the following sequence of commands with respect to a mashup with components for `homepage.example.com`, `twitter.example.com` and `spamcheck.example.com`:

```
document = fetchDocument("homepage.example.com");
updates = fetchDocument("twitter.example.com");
runScript("spamcheck.example.com");
```

This program fragment takes the data associated with the homepage, places it in the heap with fresh identifiers, and stores the root node identifier in `document`. It then extracts the Twitter updates from the Twitter component’s data, and runs the code associated with the spam checker in the same environment.

To give components a method for sharing reusable parametric code, we introduce dynamic functions. Dynamic functions are a hybrid between procedures and higher-order functions used in e.g. JavaScript. Dynamic functions are named code blocks, called bodies, with an associated parameter list of arbitrary arity. They are dynamic because the binding of the function name to a body can be changed during program execution. For example, function names may be associated with no body initially, have some body later in execution, and another body later still. This dynamic nature is needed to handle both the introduction of new functions and the replacement of existing functions by mashup components when they are executed via `runScript`. JavaScript functions are higher-order. We do not work with higher-order functions as our mashup application does not rely on this feature and, so far, higher-order functions in local reasoning require complex denotational semantics [14].

We extend our semantics with a *function table* which is a mutable mapping from function names to parameter lists and body code. We add two new statements to the language, allowing the use of this table:

`function Name(P1, ..., Pn) { Statement*; return E }`: updates the function table to associate the name `Name` with the body code between the braces along with the parameter names. This command cannot fault.

`r = Name(E1, ..., En)`: extracts the body code associated with `Name` from the function table, replaces the parameters names with the associated expressions `E1` to `En`, then behaves as this code, associating the return expression of the body code with the store variable `r`. It faults if there is no function named `Name` in the table, if the passed parameter count does not match the stored function, or if the function body faults.

The only restriction on the use of dynamic functions is that function bodies may not include function introduction statements; these may only be used at the “top level” component code. Function calls, however, may appear as statements anywhere and be (mutually) recursive. As a notation, we allow function call statements to be used as expressions; this denotes a freshly-named variable being introduced, bound to the function result, and that variable being used as the expression. To reduce reasoning verbosity, we use the standard technique of disallowing assignment to parameters within a function body.

To allow implementation details to remain private, we may want to use variables with a scope limited to a single function body. We may also want to define functions scoped to a single component. We allow both these behaviours by annotating the name with a `local` prefix. This restricts the name to the lexical scope of the containing function or component, and ensures that the name shadows any identical name from a wider scope. The implementation is largely standard and is omitted for space reasons.

**Assertion language:** As well as a heap and store, our semantics now uses a mashup and function table. Fault free use of `fetchDocument`, `runScript` and dynamic functions depend on properties of these new structures. The mashup IO commands only fault when the mashup contains no entry for the URI they are accessing. We hence extend the assertion language over program state with  $\mu(Str)$ , which is true if and only if the mashup contains an entry for `Str`. Function calls will fault when there is no entry in the function table for the function name. We introduce  $\gamma(\text{Name}(P_1, \dots, P_n))$ , which is true if and only if the function table contains a body for the given function `Name` and its parameters. Other faulting behaviours of functions will be ruled out by consistency conditions, which we give shortly.

When proving code that uses mashup IO commands, one must have information about the behaviour of the remote component being mashed in. We could force the proof to appeal to the specific data and code of the remote component, but this is highly non-modular - one would require all the implementation details when all one really wants is a specification. To enable a more modular reasoning between components, we introduce the notion of component specifications.

There are three primitive specification concepts. A *code specification* for code  $\mathbb{C}$  is a triple  $(P, Q, M)$ , where  $P$  and  $Q$  are pre- and post-conditions for  $\mathbb{C}$  and  $M$  is the modification set of  $\mathbb{C}$ : that is, the set of all non-local names modified by the code. The modification set is needed for a side-condition on the abstract frame rule that ensures we cannot frame away assertions about names the code will modify, and is standard with three additions: function introduction is considered to modify the name of the function being introduced; the modifications of a function call statement are the modifications of the function body; and for `runScript` statements, it is the set for the script code being run. For brevity, we write the empty code specification  $(\emptyset_G, \emptyset_G, \{\})$  as  $\emptyset_{\mathbb{C}}$ . A *data specification* is an assertion from the assertion language given in section 2, in which no store variables are mentioned and all identifiers are existentially quantified. A *function specification* is a finite partial function mapping every function signature (e.g. `Name(P1, ..., Pn)`) introduced by a function introduction statement in  $\mathbb{C}$  to a pair  $(C', R)$ , where  $C'$  is a code specification that describes the function body and  $R$  lists the parameters.

With these primitives, we define a *component specification* for a given component  $(\mathbb{D}, \mathbb{C})$  as a triple  $(D, C, F)$ , where  $D$  is a data specification describing  $\mathbb{D}$ ,  $C$  is a code specification for  $\mathbb{C}$ , and  $F$  is a function set specification.

Component specifications allow proofs to use remote components by depending on their published specification, rather than requiring the full details of their (possibly changing) data and code. In fact, the remote site will tend to prove a stronger specification than the published specification. The published specification is still enough to use the data and code, but this weakening allows implementation details to be hidden.

**Program reasoning:** Recall we defined mashups as finite partial functions from URIs to mashup components (code/data pairs). A *mashup specification* is a finite partial function from URIs to component specifications.

We write mashup specifications as  $\mathcal{M} = \{U_1 \mapsto (D_1, C_1, F_1), \dots, U_n \mapsto (D_n, C_n, F_n)\}$ . Given such a mashup specification  $\mathcal{M}$ , the *mashup functions* of  $\mathcal{M}$ , denoted  $\mathcal{F}(\mathcal{M})$ , is defined by

$$\mathcal{F}(\mathcal{M}) = F_1 \uplus \dots \uplus F_n$$

where

$$(F_1 \uplus F_2)(Name) \triangleq \begin{cases} F_1(Name) \iff Name \in \text{dom}(F_1) \wedge Name \notin \text{dom}(F_2) \\ F_2(Name) \iff Name \notin \text{dom}(F_1) \wedge Name \in \text{dom}(F_2) \\ F_1(Name) \iff F_2(Name) = ((P_1, Q_1, M), R) \wedge \\ \quad F_1(Name) = ((P_1, Q_1, M), R) \wedge \\ F_1(Name) \iff F_2(Name) = ((P_2, Q_2, M), R) \wedge \\ \quad (P_1 \iff P_2) \wedge (Q_1 \iff Q_2) \\ \text{Otherwise undefined} \end{cases}$$

We now extend our Hoare judgements for code to handle mashup IO commands and dynamic functions. The Hoare judgements, previously  $\{P\}C\{Q\}$ , are now parameterised with a mashup specification of the form  $\mathcal{M} \vdash \{P\}C\{Q\}$ . In the existing rules, this  $\mathcal{M}$  is carried unchanged. We use it in the mashup IO and function rules to appeal to specifications. We give a subset of the rules below, in which ignore issues of local names (they complicates the rule syntax, without being very informative).

$$\frac{\mathcal{M}(\text{StrExpr}) = (\langle \text{s}[f] \rangle_G, C, F)}{\mathcal{M} \vdash \{\mu(\text{StrExpr})\} \text{Id} = \text{fetchDocument}(\text{StrExpr}) \{\mu(\text{StrExpr}) \wedge \langle \text{sId}[f] \rangle_G\}}$$

$$\frac{\mathcal{M}(\text{StrExpr}) = (D, (P, Q, M), F)}{\mathcal{M} \vdash \{\mu(\text{StrExpr}) \wedge P\} \text{runScript}(\text{StrExpr}) \{\mu(\text{StrExpr}) \wedge Q\}}$$

$$\frac{\mathcal{F}(\mathcal{M})(\text{Name}) = ((P, Q, M), (P_1, \dots, P_n)) \quad \mathcal{M} \vdash \{P\}C\{Q\} \quad \text{mods}(C) = M}{\mathcal{M} \vdash \{\emptyset_G\} \text{function Name}(P_1, \dots, P_n) \{C\} \{\gamma(\text{Name})\}}$$

$$\frac{\mathcal{F}(\mathcal{M})(\text{Name}) = ((P, Q, M), (P_1, \dots, P_n))}{\mathcal{M} \vdash \{\gamma(\text{Name}) \wedge P[E_i/P_i]\} \mathbf{r} = \text{Name}(E_1, \dots, E_n) \{\gamma(\text{Name}) \wedge Q[E_i/P_i]\}}$$

The mashup IO rules are straightforward, simply appealing to the specifications table. The function rules are also simple; functions are looked up in the specification table, and suitable substitutions of passed expressions for parameters are made. Note that function introduction checks that the body being introduced for a name actually fulfils the promised specification.

As function bodies are checked at introduction, we should also check that mashup components fulfil their specifications. Here, we assume the check that data satisfies the data specification, and that the mashup code satisfies the code specification is made by the publishers of the specifications before they provide them. We also perform a consistency check between all the functions of all the components in a mashup, i.e. that  $F_1 \uplus \dots \uplus F_j$  is defined for all the functions.

### 3.1 Detailed example

We now have the tools to provide the implementation and associated reasoning for our Twitter example. We will build a mashup  $\mathbf{m}$  with specification  $\mathcal{M}$ .

**Twitter Update List:** The Twitter update component provides only the set of recent updates as data, and has no code. We have already given an instance of the data in section 2, so can define its component within the mashup as

$$\mathbf{m}(\text{twitter.example.com}) = (\text{See section 2 (1), skip})$$

Recall the assertion (3) in section 2, defining *twitter* as a predicate describing data in the Twitter format. As there is no code associated with Twitter, and hence no functions, we can compactly describe the public specification for the Twitter component

$$\mathcal{M}(\text{twitter.example.com}) = (\text{twitter}, \emptyset_C, \emptyset)$$

**The Spam Checking service:** We have been deliberately vague about the specifics of the spam checking component, noting only that it will provide some method of filtering spam users from the list. This mirrors our concerns as developers of the homepage - we do not mind how this is implemented, as long as it provides us with a checking method in the form of an `isSpammer` function. In figure 1, we assumed that the service would maintain a database in its data, and use other, hidden, local functions to make this service work - many other implementations are possible.

$$\mathbf{m}(\text{spamcheck.example.com}) = \left( \begin{array}{c} \mathbb{C}_1; \\ \mathbb{D}, \text{function isSpammer}(\text{name}) \{ \mathbb{C}_2 \}; \\ \mathbb{C}_2 \end{array} \right)$$

The public specification for the service mirrors this operational vagueness by giving very weak data specification, and by giving the `isSpammer` function a simple contract to fulfil. The implementation is free to behave however it wishes, as long as it provides a suitable function, does not alter the environment beyond adding this function, and does not access the web beyond its own data.

$$FSpec \triangleq \{ \text{isSpammer}(\text{name}) \mapsto ((\text{name} : \text{STR}, \text{ret} = \text{true} \vee \text{ret} = \text{false}, \emptyset), (\text{name})) \}$$

$$\mathcal{M}(\text{spamcheck.example.com}) = \left( \text{true}, \left( \begin{array}{c} \emptyset_G \wedge \mu(\text{spamcheck.example.com}), \\ \emptyset_G \wedge \mu(\text{spamcheck.example.com}) \wedge \gamma(\text{isSpammer}), \\ \{ \text{isSpammer} \} \end{array} \right), FSPEC \right)$$

**The homepage:** The homepage forms the distinguished mashup component. It uses its own data, which contains an `updatesILike` node. It downloads the Twitter data and the spam checking code, then for every update that is not spam, adds the update text as a child of the `updatesILike` node. Let the following code be  $\mathbb{C}_h$ :

```
document = fetchDocument("homepage.example.com");
updatesDoc = fetchDocument("twitter.example.com");
runScript("spamcheck.example.com");
// Find the "updatesILike" HTML node (elided)
updatesILike = ...
// Add each update
updates = getChildNodes(updatesDoc)
length = getLength(updates); i = 0;
while (i < length) {
  update = item(updates, 0);
  i = i + 1;
  // Get the update username
  userNode = getFirstChild(update);
  userName = getNodeValue(userNode);
  updateText = getSecondChild(update);
  // Spam check the username
  if (isSpammer(userName) == false) {
    newUpdate = createTextNode(getNodeValue(updateText));
    appendChild(updatesILike, newUpdate);
  } else {
    removeChild(updates, update);
  }
}
```

The associated data need only have a `updatesILike` node somewhere within it; one such document is

$$\mathbb{D}_h \triangleq \text{html}_1[\text{updatesILike}_2[\emptyset]_3]_4$$

We can then give the component for the homepage as

$$\mathbf{m}(\text{homepage.example.com}) = (\mathbb{D}_h, \mathbb{C}_h)$$

Let  $Mods$  be the set of all variables in  $\mathbb{C}_h$ . We can then specify the homepage as

$$\begin{aligned}
&htmlSpec \triangleq \text{html}[\Diamond \text{updatesI} \text{Like}[\emptyset_F]] \\
\mathcal{M}(\text{homepage.example.com}) = &\left( \begin{array}{l} htmlSpec, \\ \left( \begin{array}{l} \mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \\ \wedge \mu(\text{homepage.example.com}) \wedge \emptyset_G, \end{array} \right) \\ true, \\ Mods \\ \emptyset \end{array} \right), \end{array} \right)
\end{aligned}$$

The pre-condition for the homepage states its dependence on the Twitter and spam check services; other than that, it needs nothing. The post-condition is just true. This is enough to prove we have a fault free mashup; that is, the only resources required are those given in the pre-condition. If, however, we expected our homepage to be “mashed into” other pages, a stronger post-condition would make our service much more useful. The proof gives a post-condition that describes the homepage data, and how the only change within it is the addition of Twitter updates. It also shows how ever update from the Twitter updates list has been processed.

**Proof of the client code:** We give the proof of the homepage code with respect to  $\mathcal{M}$  in figure 3, under the assumption that the other components within the mashup specification have been proven with respect to their specifications. We do not give every Hoare rule application, providing instead assertions describing the intermediary states and the loop invariant.

**Limitations of the example:** Although realistic, this example has not exercised the full range of our system. There was no dynamic function replacement and due to space, the data reasoned over has been relatively simple. Larger examples are more verbose, but rarely require significantly more ingenuity. Our experiments so far suggest that most mashups will fall under the pattern of our Twitter example.

## 4 Conclusions

We have presented a simple formulation of mashup programs and a reasoning system for specifying properties about such programs. With our reasoning, we can prove that mashups are fault-free and provide specifications for code that are analogous to XML schema for data. We have illustrated our work using a realistic and non-trivial example, giving hope that these techniques are not purely abstract but should be useful in removing actual software faults.

As web data can be described with XML schema, we have shown how mashup code can be described with our code specifications. Each component within a mashup can be independently verified, and the resulting specifications can be published to other programmers who reuse the component with confidence. This result is *mashup safety*. If all components in a mashup are proven with respect to the same set of specifications, then they cannot fault: that is, they cannot access a DOM node that does not exist, attempt a prohibited DOM operation, access a web site that is not available, or call a function that has not been introduced. This result has other benefits. For example, the scope of any proven mashup is well understood, so firewalls can be configured so that they do not accidentally block a component of a mashup that it not obviously seen to be used.

This work represents the first steps towards reasoning about larger web programs, a path that should lead us to reason about several web technologies. We continue to analyse our techniques with respect to real web examples, and hope to perform a more complete statistical analysis of mashup usage patterns on the web. We are actively pursuing local reasoning for JavaScript. Noting the verbose, but relatively mechanical nature of the proof in figure 2, we are investigating automation for our reasoning technique, extending the work of jStar [15]. We also plan to apply our techniques to further examples and problems in the mashup area. One intriguing area is *mashup security*. For example, if your bank wanted to “mash in” a component to their online banking system, how can they be sure the component is not surreptitiously altering the document?

**Acknowledgements** We thank EPSRC, the Royal Academy of Engineering and Microsoft Research Cambridge for their financial support.

```

{ $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \emptyset_G$ }
document = fetchDocument("homepage.example.com");
{  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge$ 
   $\langle \text{html}_{\text{document}} [\diamond \text{updatesILike}[\emptyset_F]] \rangle_G$  }

twitterDoc = fetchDocument("twitter.example.com/twitter");
{  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge$ 
   $\langle \text{html}_{\text{document}} [\diamond \text{updatesILike}[\emptyset_F]] \oplus \text{twitter}_{\text{twitterDoc}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update}] \rangle_G$  }

runScript("spamcheck.example.com");
{  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \gamma(\text{isSpammer}) \wedge$ 
   $\langle \text{html}_{\text{document}} [\diamond \text{updatesILike}[\emptyset_F]] \oplus \text{twitter}_{\text{twitterDoc}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update}] \rangle_G$  }

updatesILike = ...
{  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \gamma(\text{isSpammer}) \wedge$ 
   $\langle \text{html}_{\text{document}} [\diamond \text{updatesILike}_{\text{updatesILike}}[\emptyset_F]] \oplus$ 
   $\langle \text{twitter}_{\text{twitterDoc}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update}] \rangle_G$  }

updates= getChildNodes(twitterDoc)
length = getLength(updates); i = 0;
{  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \gamma(\text{isSpammer}) \wedge$ 
   $\langle \exists F. \langle \text{html}_{\text{document}} [\diamond \text{updatesILike}_{\text{updatesILike}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \# \text{text}]] \oplus$ 
   $\langle \text{twitter}_{\text{twitterDoc}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update} \wedge F \wedge \text{len}(F) = \text{length}] \rangle_G$  }
while (i < length) {
  update = item(updates, i);
  {  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \gamma(\text{isSpammer}) \wedge$ 
     $\langle \exists F_1, F_2, X. \langle \text{html}_{\text{document}} [\diamond \text{updatesILike}_{\text{updatesILike}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \# \text{text}]] \oplus$ 
     $\langle \text{twitter}_{\text{twitterDoc}} \left[ \begin{array}{l} \square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update} \\ \wedge (F_1 \otimes \text{update}_{\text{update}} [\text{user}[\# \text{text}[X]] \otimes \# \text{text}] \otimes \\ F_2 \wedge \text{len}(F_1) = i \wedge \text{len}(F_2) = \text{length} - i - 1) \end{array} \right] \rangle_G$  }
    i = i + 1;
    userNode = getFirstChild(update);
    userName = getNodeValue(userNode);
    updateText = getSecondChild(update);
    {  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \gamma(\text{isSpammer}) \wedge$ 
       $\langle \exists F_1, F_2, X. \langle \text{html}_{\text{document}} [\diamond \text{updatesILike}_{\text{updatesILike}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \# \text{text}]] \oplus$ 
       $\langle \text{twitter}_{\text{twitterDoc}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update} \wedge (F_1 \otimes \text{update}_{\text{update}} [$ 
       $\text{user}_{\text{userNode}} [\# \text{text}[X \wedge X = \text{userName}]] \otimes \# \text{text}_{\text{updateText}}$ 
       $\rangle \otimes F_2 \wedge \text{len}(F_1) = i - 1 \wedge \text{len}(F_2) = \text{length} - i] \rangle_G$ 
      if (isSpammer(userName) == false) {
        newUpdate = createTextNode(getNodeValue(updateText));
        appendChild(updatesILike, newUpdate);
      }
      else { skip; }
    }
  }
  {  $\mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \mu(\text{homepage.example.com}) \wedge \gamma(\text{isSpammer}) \wedge$ 
     $\langle \text{html}_{\text{document}} [\diamond \text{updatesILike}_{\text{updatesILike}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \# \text{text}]] \oplus$ 
     $\langle \text{twitter}_{\text{twitterDoc}} [\square_{\otimes} \langle \text{true}_T \rangle_F \implies \text{update}] \rangle_G$  }
}
{true}

```

**Fig. 3.** Proof of our Twitter enabled homepage

# Bibliography

- [1] O’Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. (2001)
- [2] Gardner, P.A., Smith, G.D., Wheelhouse, M.J., Zarfaty, U.D.: Local Hoare reasoning about DOM. In: Proceedings, PODS’08. (2008)
- [3] Smith, G.D.: PhD thesis. In preparation
- [4] Thiemann, P.: A type safe DOM API. In: Proceedings DBPL, 169–183. (2005)
- [5] Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Proceedings, APLAS 2008. LNCS (2008)
- [6] Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*
- [7] Gregor Richards, Sylvain Lebesne, B.B.J.V.: An analysis of the dynamic behavior of javascript program
- [8] <http://www.adsafe.org/>: Adsafe
- [9] M.S. Miller, M. Samuel, B.L.I.A., Stay, M.: Caja: Safe active content in sanitized JavaScript. In: A Google research project. (2008)
- [10] Jackson, C., Wang, H.: Subspace: Secure cross-domain communication for web mashups
- [11] C. Reis, S.G., Levey, H.: Architectural principles for safe web programs. (2009)
- [12] Klose, K., Ostermann, K.: Modular logic metaprogramming. (2009)
- [13] : Document Object Model Level 1 specification
- [14] Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested hoare triples and frame rules for higher-order store. LNCS, Springer (2009)
- [15] Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: OOPSLA, ACM (2008)
- [16] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings, LICS 2002). (2002)
- [17] Gardner, P., Smith, G., Wheelhouse, M.J., Zarfaty, U.: DOM: Towards a formal specification. In: PLAN-X. (2008)
- [18] Hoare: An axiomatic basis for computer programming. *CACM: Communications of the ACM* **26** (1983)
- [19] Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. 32nd POPL’2005, ACM SIGPLAN Notices **40**(1) (2005)
- [20] World Wide Web Consortium: HTML 4.01 Specification. (December 1999)
- [21] Association, E.C.M.: ECMAScript Language Specification. (June 1997)
- [22] Gardner, P., Smith, G., Wright, A.: Local reasoning about mashups : Technical report. (2010)