DOM: Specification and Client Reasoning

Azalea Raad, José Fragoso Santos and Philippa Gardner

Imperial College London

Abstract. We present an axiomatic specification of a key fragment of DOM using structural separation logic. This specification allows us to develop modular reasoning about client programs that call the DOM.

1 Introduction

The behaviour of JavaScript programs executed in the browser is complex. Such programs manipulate a heap maintained by the browser and call a wide range of APIs via specific objects in this heap. The most notable of these is the Document Object Model (DOM) API and the DOM document object, which are used to represent and manipulate the web page. JavaScript programs must run uniformly across all browsers. As such, the English standards of JavaScript and DOM are rather rigorous and are followed closely by browser vendors. While there has been work on formal specifications of JavaScript [14], including mechanised specifications [4], and some work on the formal specification of DOM [9,22] and on the verification of JavaScript programs [7], we are not aware of any work on the verification of JavaScript programs that call the DOM.

The W3C DOM standard [1] describes an XML update library used by all browsers. This English standard is written in an axiomatic style that lends itself well to formalisation. The first formal axiomatic DOM specification has been given in [9,22], using context logic (CL) [6,5], which extends ideas from separation logic (SL) [19] to complex data structures. However, this work has several shortcomings. First, it is not simple to integrate SL reasoning about e.g. C [19], Java [16] and JavaScript [7] with the DOM specifications in CL. The work in [9,22] explores the verification of simple client programs manipulating a variable store and calling the DOM. It does not verify clients manipulating a standard program heap. Second, this specification does not always allow *compositional* client-side reasoning. Finally, this specification makes simplifying choices (e.g. with live collections), and does not always remain faithful to the standard.

We present a faithful axiomatic specification of a key fragment of the DOM and verify substantial client programs, using structural separation logic (SSL) introduced in [25,8]. SSL provides fine-grained reasoning about complex data structures. The SSL assertion language contains the commutative separating conjunction ($*$), as in SL, that serves to split the DOM tree into smaller subtrees. By contrast, the CL assertion language contains the non-commutative separating application ($\bullet$), that splits the DOM tree into a tree context with a hole applied to a partial DOM tree. These two operators are not compatible with each other.

In particular, the integration of the CL DOM specification with an SL-based program logic involves extending the program logic to include a frame rule for the separating application. By contrast, the integration of our SSL DOM specification with an SL-based program logic requires no extensions. We can reason about DOM client programs written in e.g. C, Java and JavaScript, by simply using a combination of the appropriate SL-based program logic for reasoning about the particular programming language and our DOM axioms. We illustrate this by verifying several realistic ad-blocker client programs written in JavaScript, using the program logic of [7]. Our reasoning abstracts the complexities of JavaScript, simply using standard SL assertions, an abstract variable store predicate, and JavaScript heap assertions. It is thus straightforward to transfer our ideas to other languages, as we show in §3.

As the authors noted in [9,22], CL does not always allow for *local* reasoning. As we demonstrate in§2, it also does not provide *compositional* reasoning. In contrast, SSL provides both local and compositional client reasoning. We demonstrate this by presenting a simple client program which can be specified using a *single* SSL triple whose precondition captures its intuitive footprint, compared to *six* CL triples, whose preconditions are substantially larger than the footprint.

The DOM English standard [1] is written in an axiomatic style, allowing for a straightforward comparison of our formal axiomatic specification with the standard. A typical way to justify an axiomatic specification of a library is to compare it against an operational semantics, as in [9,22,25] for DOM. However, this approach seems unsuitable as it involves inventing an operational semantics for the library, even though the standard is written in an axiomatic style. Instead, we justify our specification with respect to a reference implementation that can be independently tested. In [17] we present a JavaScript implementation of our DOM fragment, and prove its correctness with respect to our specification.

Related work There has been much work on simple models of semi-structured data, following the spirit of DOM, such as [6,2,3] (axiomatic, program logic) and [20] (operational, information flow). We do not detail this work here. Instead, we concentrate on axiomatic and operational models, with a primary focus on DOM. Smith et al. developed an axiomatic specification of the DOM [9,22] in CL [6,5], as discussed above. Others have also studied operational models of DOM. Lerner et al. were the first to formalise the DOM event model [13]. This model is executable and can be used as an oracle for testing browser compliance with the standard. Unlike our work, this model was not designed for proving functional properties of client programs, but rather meta-properties of the DOM itself. The main focus of this work is the event dispatch model in DOM. Rajani et al. [18] have developed an operational model for DOM events and live collections, in order to study information flow. We aim to study DOM events in the future.

There has been much work on type analysis for client programs calling the DOM. Thiemann [24] developed a type system for establishing safety properties of DOM client programs written in Java. He extended the Java type system of [10] with recursion and polymorphism, and used this extension to specify the DOM data structures and operations. Later, Jensen et al. added DOM types

to JavaScript [12,21,11], developing a flow sensitive type analysis tool TAJS. They used DOM types to reason about control and data flow in JavaScript applications that interact with the DOM. Recently, Park et al. developed a framework for statically analysing JavaScript web applications that interact with the DOM [15]. As with TAJS, this framework uses configurable DOM abstraction models. However, the proposed models are significantly more fine-grained than those of TAJS in that they can precisely describe the structure of DOM trees whereas TAJS simply treats them as black boxes. In [23], Swamy et al. translate JavaScript to a typed language and type the DOM operations. The DOM types are intentionally restrictive to simplify client analysis (e.g. modelling live collections as iterators in [23]). In contrast, there has been little work on the verification of programs calling the DOM. Smith et al. [9,22] look at simple client programs which manipulate the variable store and the DOM. However, their reasoning is not compositional, as previously discussed and formally justified in §2.

Outline In §2, we summarise our contributions. In §3, we present our DOM specification and describe how our specification may be integrated with an arbitrary SL-based program logic. In §4, we verify a JavaScript ad-blocker client program which calls the DOM, and we finish with concluding remarks.

2 Overview

2.1 A Formal DOM Specification

The W3C DOM standard [1] is presented in an object-oriented (OO) and language-independent fashion. It consists of a set of interfaces describing the fields and methods exposed by each DOM datatype. A DOM object is a tree comprising a collection of *node* objects. DOM defines twelve specialised node types. As our goal is to present our specification methodology, we focus on an expressive fragment of DOM Core Level 1 (CL1) that allows us to create, update, and traverse DOM documents. We thus model the four most commonly used node types: *document*, *element*, *text* and *attribute* nodes. Additionally, we model *live collections of nodes* such as the *NodeList* interface in DOM CL1-4 (discussed in §3.5). Our fragment underpins DOM Core Levels 1-4. As shown in [22], it is straightforward to extend this fragment to the full DOM CL1 without adding to the complexity of the underlying program logic. It will be necessary to extend the program logic as we consider additional features in the higher levels of the standard (e.g. DOM events). However, these features will not affect the fragment specified here. We proceed with an account of our DOM fragment, hereafter simply called DOM.

DOM nodes Each node in DOM is associated with a *type*, a *name*, an optional *value*, and information about its surroundings (e.g. its parent, siblings, etc.). Given the OO spirit of the standard, each node object is uniquely identified by its reference. To capture this more abstractly (and admit non-OO implementations), we associate each node with a unique *node identifier*. As mentioned earlier, the standard defines twelve different node types of which we model the following

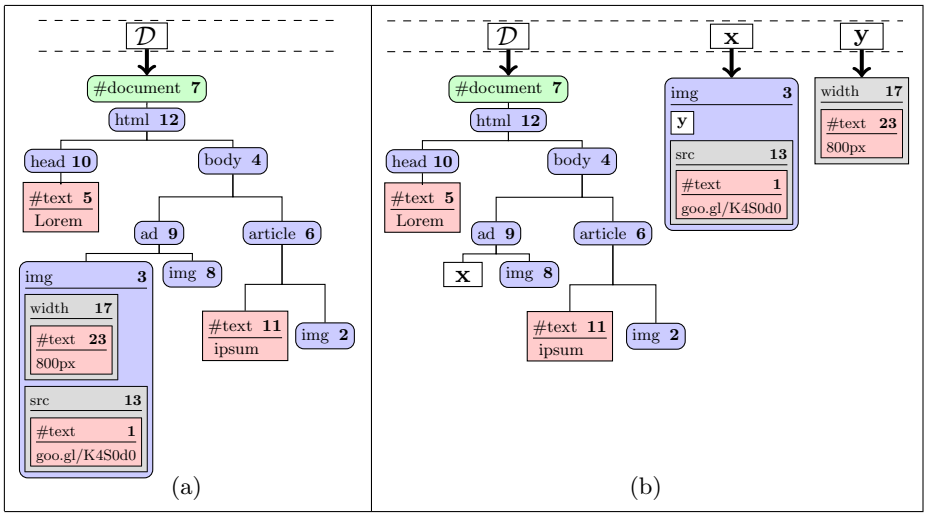

Fig. 1: A complete DOM heap (a); same DOM heap after abstract allocation (b)

four. *Document nodes* represent entire DOM objects. Each DOM object contains exactly one document node, named #document, with no value and at most a single child, referred to as the *document element*. In Fig. 1a, the document node is the node with identifier 7 (with document element 12). *Text nodes* (named #text) represent the textual content of the document. They have no children and may have arbitrary strings as their values. In Fig. 1a, node 5 is a text node with string data "Lorem". *Element nodes* structure the contents of a DOM object. They have arbitrary names (not containing the '#' character), no values and an arbitrary number of text and element nodes as their children. In Fig. 1a, node 12 is an element node with name "html" and two children with identifiers 10 and 4. *Attribute nodes* store information about the element nodes to which they are attached. The attributes of an element must have unique names. Attribute nodes may have arbitrary names (not containing the '#' character) and an arbitrary number of text nodes as their children. The value of an attribute node is given by concatenating the values of its children. In Fig. 1a, the element node with identifier 3 has two attributes: one with name "width", identifier 17, and value "800px" (i.e. the value of text node 23); and another with name "src", identifier 13, and value "goo.gl/K4S0d0" (i.e. the value of text node 1).

DOM operations The complete set of DOM operations and their axioms are given in [17]. In §3, we present the axioms for the operations used in the examples of this paper. Here, we describe the `n.getAttribute(s)` and `n.setAttribute(s,v)` operations and their axioms to give an intuitive account of SSL. The `n.getAttribute(s)` operation inspects the attributes of element node `n`. It returns the value of the attribute named `s` if it exists, or the empty string otherwise. For instance, given the DOM tree of Fig. 1a, when variable `n`

holds value 3 (the element node named "img", placed as the left child of node "ad"), and `s` holds "src", then `r = n.getAttribute(s)` yields `r`="goo.gl/K4S0d0".

Intuitively, the footprint of `n.getAttribute(s)` is limited to the element node `n` and its "src" attribute. To describe this footprint minimally, we need to split the element node at `n` away from the larger surrounding DOM tree. To do this, we introduce *abstract DOM heaps* that store abstract tree fragments. For instance, Fig. 1a contains an abstract DOM heap with one cell at address $\mathcal{D}$ and a complete abstract DOM tree as its value. It is abstract in that it hides the details of how a DOM tree might be concretely represented in a machine. Abstract heaps allow for their data to be split by imposing additional instrumentation using *abstract addresses*. Such splitting is illustrated by the transition from Fig. 1a to Fig. 1b. The heap in Fig. 1a contains a complete tree at address $\mathcal{D}$. This tree can be split using *abstract allocation* to obtain the heap in Fig. 1b with the subtree at node 3 at a fresh, fictional *abstract cell* $\mathbf{x}$, and an incomplete tree at $\mathcal{D}$ with a *context hole* $\mathbf{x}$ indicating the position to which the subtree will return. Since we are only interested in the attribute named "src", we can use abstract allocation again to split away the other unwanted attribute ("width") and place it at a fresh abstract cell $\mathbf{y}$ as illustrated in Fig. 1b. The subtree at node 3 and its "src" attribute correspond to the intuitive footprint of `n.getAttribute(s)`. Once the `getAttribute` operation is complete, we can join the tree back together through *abstract deallocation*, as in the transition from Fig. 1b to 1a.

Using SSL [25], we develop *local* specifications of DOM operations that only touch the intuitive footprints of the operations. The assertion language comprises *DOM assertions* that describe abstract DOM heaps. For example, the DOM assertion $\alpha \mapsto \mathrm{img}_3[\beta \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\text{goo.gl/K4S0d0}]], \varnothing]$ describes the abstract heap cell at $\mathbf{x}$ in Fig. 1b, where α and β denote logical variables corresponding to abstract addresses $\mathbf{x}$ and $\mathbf{y}$, respectively. It states that the heap cell at abstract logical address α holds an "img" element with identifier 3, no children ($\varnothing$) and a set of attributes described by $\beta \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\text{goo.gl/K4S0d0}]]$, which contains a "src" attribute (with identifier 13 and value "goo.gl/K4S0d0") and other attributes to be found at abstract logical address β. The attributes of a node are grouped by the commutative $\odot$ operator. When we are only interested in the value of an attribute, we can write an assertion that is agnostic to the shape of the text content under the attribute. For instance, we can write $\alpha \mapsto \mathrm{img}_3[\beta \odot \mathrm{src}_{13}[\mathrm{T}], \varnothing] * \mathsf{val}(\mathrm{T}, \text{goo.gl/K4S0d0})$ to state that attribute 13 contains some text content described by logical variable T, and that the value of T (i.e. the value of the attribute) is "goo.gl/K4S0d0". Assertion $\mathsf{val}(\mathrm{T}, \text{goo.gl/K4S0d0})$ is *pure* in that it contains no resources and merely describes the string value of T.

Using SSL triples, we can now locally specify `r = n.getAttribute(s)` as[1]:

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : -) \\ * \, \alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma] \\ * \, \mathsf{val}(\mathrm{T}, \mathrm{S}'')\end{array}\right\} \ \mathtt{r = n.getAttribute(s)} \ \left\{\begin{array}{l}\mathsf{store}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{S}'') \\ * \, \alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma] \\ * \, \mathsf{val}(\mathrm{T}, \mathrm{S}'')\end{array}\right\} \quad (1)$$

[1] It is possible to combine multiple cases into one by rewriting the pre- and postconditions as a disjunction of the cases and using logical variables to track each case. For clarity, we opt to write each case separately.

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:-)\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]*\mathsf{out}(\mathrm{A},\mathrm{S})\end{array}\right\}\ \mathtt{r=n.getAttribute(s)}\ \left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\text{“”})\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]*\mathsf{out}(\mathrm{A},\mathrm{S})\end{array}\right\}\quad(2)$$

SSL triples have a fault-avoiding, partial-correctness interpretation as in other separation logics: if an abstract DOM heap satisfies the precondition then either the operation does not terminate, or the operation terminates and the resulting state will satisfy the postcondition. Axiom (1) captures the case when `n` contains an attribute named `s`; axiom (2) when `n` has no such attribute. The precondition of (1) contains three assertions. Assertion $\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:-)$ describes a variable store where program variables `n`, `s` and `r` have logical values N, S and an arbitrary value $(-)$, respectively.[2] Assertion $\alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}],\gamma]$ describes an abstract DOM heap cell at the logical abstract address α containing the subtree described by assertion $\mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}],\gamma]$. This assertion describes a subtree with a single element node with identifier N and name S′. Its children have been framed off, leaving behind the context hole γ (using abstract allocation as in the transition from Fig. 1a to 1b, then framing off the cell at γ). It has an attribute named S with identifier M and text content T, plus (potentially) other attributes that have been framed off, leaving behind the context hole β. This framing off of the children and attributes other than `s` captures the intuition that the footprint of `n.getAttribute(s)` is limited to element `n` and attribute `s`. Lastly, assertion $\mathsf{val}(\mathrm{T},\mathrm{S}'')$ states that the value of text content T is S″. The postcondition of (1) declares that the subtree remains the same and that the value of `r` in the variable store is updated to S″, i.e. the value of the attribute named S.

The precondition of (2) contains the assertion $\alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]$ where, this time, the attributes of the element node identified by N are described by the logical variable A. With the precondition of (1), all other attributes can be framed off leaving context hole β. With the precondition of (2) however, the attributes are part of the intuitive footprint since we must check the absence of an attribute named S. This is captured by the $\mathsf{out}(\mathrm{A},\mathrm{S})$ assertion. The postcondition of (2) declares that the subtree remains the same and the value of `r` in the variable store is updated to the empty string “”, as mandated by the English specification.

The `n.setAttribute(s,v)` operation inspects the attributes of element node `n`. It then sets the value of the attribute named `s` to `v` if such an attribute exists (3). Otherwise, it creates a new attribute named `s` with value `v` and attaches it to node `n` (4). We can specify this English description as[1]:

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}],\gamma]\\ *\ \delta\mapsto \varnothing_g\end{array}\right\}\ \mathtt{n.setAttribute(s,v)}\ \left\{\begin{array}{l}\exists \mathrm{R}.\ \mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}'']],\gamma]\\ *\ \delta\mapsto \mathrm{T}\end{array}\right\}\quad(3)$$

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]*\mathsf{out}(\mathrm{A},\mathrm{S})\end{array}\right\}\ \mathtt{n.setAttribute(s,v)}\ \left\{\begin{array}{l}\exists \mathrm{M},\mathrm{R}.\ \mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A}\odot \mathrm{S}_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}'']],\gamma]\end{array}\right\}\quad(4)$$

Recall that attribute nodes may have an arbitrary number of text nodes as their children where the concatenated values of the text nodes denotes the value of the

[2] Since DOM may be called by different client programs written in different languages, store denotes a *black-box* predicate that can be instantiated to describe a variable store in the client language. Here, we instantiate it as the JavaScript variable store.

attribute. As such, when `n` contains an attribute named `s`, its value is set to `v` by removing the existing children (text nodes) of `s`, creating a new text node with value `v` and attaching it to `s` (axiom 3). What is then to happen to the removed children of `s`? In DOM, nodes are not disposed of: whenever a node is removed, it is no longer a part of the DOM tree but still exists in memory. To model this, we associate the document object with a *grove* designating a space for the removed nodes. The $\delta \mapsto \varnothing_g$ assertion in the precondition of (3) simply reserves an empty spot ($\varnothing_g$) in the grove. In the postcondition the removed children of `s` (i.e. T) are moved to the grove. Similarly, when `n` does not contain an attribute named `s`, a new attribute named `s` is created and attached to `n`. The value of `s` is set to `v` by creating a new text node with value `v` and attaching it to `s` (axiom 4).

Comparison with existing work [9,22] In contrast to the *commutative* separating conjunction $*$ in SSL, context logic (CL) and multi-holed context logic (MCL) use a *non-commutative separating application* $\bullet$ to split the DOM tree structure. For instance, the $C \bullet_\alpha P$ formula describes a tree that can be split into a context tree C with hole α and a subtree P to be applied to the context hole. The application is not commutative; it does not make sense to apply a context to a tree. In [9,22], the authors noted that the `appendChild` axiom was not *local*, as it required more than the intuitive footprint of the operation. What they did not observe was that CL client reasoning is not *compositional*. Consider a program $\mathbb{C}$ that copies the value of the "src" attribute in element `p` to that of `q`:

$$\mathbb{C} \triangleq \texttt{s=p.getAttribute("src"); q.setAttribute("src",s)}$$

Let us assume that `p` contains a "src" attribute while `q` does not. Using SSL, we can specify $\mathbb{C}$ as follows, where $S \triangleq \mathsf{store}(\texttt{p}{:}\mathrm{P},\texttt{q}{:}\mathrm{Q},\texttt{s}{:}{-}) * \mathsf{val}(\mathrm{T},\mathrm{S}_1) * \mathsf{out}(\mathrm{A},\mathrm{S})$, $P \triangleq \mathrm{S_P}[\gamma_1 \odot \mathrm{src_N}[\mathrm{T}],\mathrm{F}_1]$, $Q \triangleq \mathrm{S'_Q}[\mathrm{A},\mathrm{F}_2]$ and $Q' \triangleq \mathrm{S'_Q}[\mathrm{A} \odot \mathrm{S_M}[\#\mathrm{text_R}[\mathrm{S}_1]],\mathrm{F}_2]$:

$$\{S * \alpha \mapsto P * \beta \mapsto Q\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * \alpha \mapsto P * \beta \mapsto Q'\} \tag{5}$$

Observe that the P and Q elements may be in one of three orientations with respect to one another: i) P and Q are not related and describe disjoint subtrees; ii) Q is an ancestor of P; and iii) P is an ancestor of Q. All three orientations are captured by (5). In contrast, using MCL (adapted to our notation) $\mathbb{C}$ is specified as follows where i-iii correspond to the three orientations above.

i) $\{S * ((C \bullet_\alpha P) \bullet_\beta Q)\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * ((C \bullet_\alpha P) \bullet_\beta Q')\}$
ii) $\{S * (Q \bullet_\alpha P)\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * (Q' \bullet_\alpha P)\}$ iii) $\{S * (P \bullet_\alpha Q)\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * (P \bullet_\alpha Q')\}$

When P and Q are not related, the precondition of (i) states that the DOM tree can be split into a subtree with top node Q, and a tree context with hole variable β satisfying the $C \bullet_\alpha P$ formula. This context itself can be split into a subcontext with top node P and a context C with hole α. The postcondition of (i) states that Q is extended with a "src" attribute, and the context $C \bullet_\alpha P$ remains unchanged. This specification is not *local* in that it is larger than the intuitive footprint of $\mathbb{C}$. The only parts of the tree required by $\mathbb{C}$ are the two elements P and Q. However, the precondition in (i) also requires the surrounding *linking* context C: to assert

that P and Q are not related (P is not an ancestor of Q and vice versa), we must appeal to a linking context C that is an ancestor of both P and Q. This results in a significant overapproximation of the footprint. As either C or P, but not both, may contain context hole β, (i) includes the behaviour of (iii), which can thus be omitted. We have included it as it is more local.

More significantly however, due to the non-commutativity of $\bullet$ we need to specify (ii) and (iii) separately. Therefore, the number of CL axioms of a client program may grow rapidly as its footprint grows. Consider the program $\mathbb{C}'$ below:

$$\mathbb{C}' \triangleq \texttt{s=p.getAttribute("src"); s'=r.getAttribute("src");}\\ \texttt{q.setAttribute("src",s+s')}$$

with its larger footprint given by the distinct `p`, `q`, `r`. When `p` and `q` contain a "src" attribute and `r` does not, we can specify $\mathbb{C}'$ in SSL with *one* axiom similar to (5). By contrast, when specifying $\mathbb{C}'$ in MCL, not only is locality compromised in cases analogous to (i) above, but we need *eight* separate specifications. Forgoing locality, as described above, we still require *six* specifications. This example demonstrates that CL reasoning is not compositional for client programs.

2.2 Verifying JavaScript Programs that Call the DOM

We demonstrate how to use our DOM specification to reason about client programs that call the DOM. Our DOM specification is agnostic to the choice of client programming language. In contrast to previous work [9,22], our DOM specification integrates simply with any SL-based program logic such as those for Java [16] and JavaScript [7]. Here, we choose to reason about JavaScript client programs.

We study a JavaScript *image sanitiser* that sanitises the "src" attribute of an element node by replacing its value with a trusted URL if the value is blacklisted. To determine whether or not a value is blacklisted, a remote database is queried. The results of successful lookups are stored in a local cache to minimise the number of queries. In §4, we use this sanitiser to implement an *ad blocker* that filters untrusted contents of a web page. The code of this sanitiser, `sanitiseImg`, is given in Fig. 2. It inspects the `img` element node for its "src" attribute (line 2). When such an attribute exists (line 3), it consults the local cache (`cache`) to check whether its value (`url`) is blacklisted (line 4). If so, it changes its value to the trusted `cat` value. If the cache lookup is unsuccessful (line 6), the database is queried by the `isBlackListed` call (line 7). If the value is deemed blacklisted (line 8), the value of "src" is set to the trusted `cat` value (line 9), and the local cache is updated to store the lookup result (line 10). Observe that `sanitiseImg` does not use JavaScript-specific constructs (e.g. eval) and simply appeals to the standard language constructs of a while language. As such, it is straightforward to transform this proof to verify `sanitiseImg` written in e.g. C and Java.

The behaviour of `sanitiseImg` is specified in Fig. 2. The specifications in (6)-(9) capture different cases of the code as follows: in (6) `img` has no "src" attribute (i.e. the conditional of line 3 fails); in (7) the value of "src" is blacklisted in the local cache (line 5); in (8) the value is blacklisted and the cache has no

$\mathsf{st} \triangleq \mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}-,\mathtt{isB}{:}-)$ $\quad P_{\mathsf{out}} \triangleq \alpha \mapsto \mathrm{S}_\mathrm{N}[\mathrm{A},\gamma] * \mathsf{out}(\mathrm{A},\text{“src”})$

$P \triangleq \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{src}_\mathrm{M}[\mathrm{T}],\gamma] * \mathsf{val}(\mathrm{T},\mathrm{S}_1) * \delta \mapsto \varnothing_g$ $\quad Q \triangleq \exists \mathrm{R}.\, \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{src}_\mathrm{M}[\#\mathrm{text}_\mathrm{R}[\mathrm{S}_2]],\gamma] * \delta \mapsto \mathrm{T}$

$\{\mathsf{st} * P_{\mathsf{out}}\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * P_{\mathsf{out}}\}$ (6)

$\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$ (7)

$\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$ (8)

$\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \neg\mathsf{isB}(\mathrm{S}_1)\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \neg\mathsf{isB}(\mathrm{S}_1)\}$ (9)

$\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}-) * \mathsf{isB}(\mathrm{S}_1)\}$ `isB=isBlackListed(url)` $\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * \mathsf{isB}(\mathrm{S}_1)\}$

$\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}-) * \neg\mathsf{isB}(\mathrm{S}_1)\}$ `isB=isBlackListed(url)` $\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}0) * \neg\mathsf{isB}(\mathrm{S}_1)\}$

$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}-,\mathtt{isB}{:}-) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$

```
1.  sanitiseImg(img,cat) ≜ {
2.    url=img.getAttribute("src");
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}-) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
3.    if (url){ // img has an attribute named “src”
4.      isB=cache.url;
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}0) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
5.      if (isB){img.setAttribute("src",cat)}  // url is in cache (thus blacklisted)
6.      else {  // url is not in cache
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}0) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
7.        isB=isBlackListed(url);
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
8.        if (isB){  // url is blacklisted
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
9.          img.setAttribute("src",cat);
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
10.         cache.url = 1
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$
```
11. } } } }
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}-,\mathtt{isB}{:}-) * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$

Fig. 2: The specifications of `sanitiseImg` (above); a proof sketch of (8) (below)

record of it (lines 9-10); and in (9) the value is not blacklisted and the cache has no record of it (i.e. the conditional of 8 fails). We focus on (8) here; the remaining ones are analogous. The precondition of (8) consists of four assertions: the st captures the values of program variables; the P describes an element with an attribute named “src” and value S_1; the $(\mathrm{C},\mathrm{S}_1) \mapsto 0$ asserts that the S_1 field of cache C holds value 0 (i.e. value S_1 may or may not be blacklisted but the cache has no record of it); and $\mathsf{isB}(\mathrm{S}_1)$ states that S_1 is blacklisted. This last assertion is used in the `isBlackListed` call of line 7 with its behaviour as specified in Fig. 2. A proof sketch of specification (8) is given in Fig. 2. At each proof point, we have highlighted the effect of the preceding command, where applicable.

3 A Formal DOM Specification

We give our formal axiomatic specification of DOM, comprising the DOM model in §3.1 eliding some details about DOM live collections until §3.5, the DOM

assertions in §3.2, the framework for reasoning about DOM client programs in §3.3, the DOM axioms in §3.4, and DOM live collections in §3.5.

3.1 DOM Model

We model *DOM heaps* (e.g. Fig. 1) as mappings from addresses to DOM data. To this end, we assume a countably infinite set of *identifiers*, $n \in \text{ID}$, a designated *document identifier* associated with the document object, $d \in \text{ID}$, a countably infinite set of *abstract addresses*, $\mathbf{x} \in \text{AADD}$, and a designated *document address* $\mathcal{D}$, where the sets ID, AADD and $\{\mathcal{D}\}$ are pairwise disjoint.

DOM data DOM nodes are the building blocks of DOM data. Formally, we write: i) $\#\text{text}_n[s]_{fs}$ for the text node with identifier n and text data s; ii) $s_n[\mathbf{a}, \mathbf{f}]^{ts}_{fs}$ for the element node with identifier n, tag name s, attribute set $\mathbf{a}$, and children $\mathbf{f}$; iii) $s_n[\mathbf{tf}]_{ts}$ for the attribute node with identifier n, name s, and children $\mathbf{tf}$; and iv) $\#\text{doc}_d[\mathbf{e}]^{ts}_{fs}$ & $\mathbf{g}$ for the document object with the designated identifier d, document element $\mathbf{e}$ (or $\varnothing_e$ for no document element) and *grove* $\mathbf{g}$, ignoring the *fs* and *ts* for now. DOM nodes can be grouped into attribute sets, forests, groves, and text forests, respectively ranged over by $\mathbf{a}$, $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{tf}$. An attribute set represents the attribute nodes associated with an element node and is modelled as an *unordered*, possibly empty collection of attribute nodes. A forest represents the children of an element node, modelled as an *ordered*, possibly empty collection of element and text nodes. A grove is where the orphaned nodes are stored. In DOM, nodes are never disposed of and whenever a node is removed from the document, it is moved to the grove. The grove is also where newly created nodes are placed. The document object is thus associated with a grove, modelled as an *unordered*, possibly empty collection of text, element and attribute nodes. A text forest represents the children of an attribute node, modelled as an *ordered*, possibly empty collection of text nodes. We associate each node with a set of *forest listeners*, *fs*; we further associate element and document nodes with a set of *tag listeners*, *ts*. We delay the motivation for these listeners until §3.5 when we model live collections. DOM data may be either incomplete with context holes (e.g. $\mathbf{x}$), or complete with no context holes. Notationally, data written in **bold** may contain context holes; regular font indicates the absence of context holes.

Definition 1. *The sets of* strings $s \in \mathbb{S}$, texts $t \in \mathbb{T}$, elements $\mathbf{e} \in \mathbb{E}$, documents $\mathbf{doc} \in \mathbb{D}$, attribute sets $\mathbf{a} \in \mathbb{A}$, forests $\mathbf{f} \in \mathbb{F}$, groves $\mathbf{g} \in \mathbb{G}$, *and* text forests $\mathbf{tf} \in \mathbb{TF}$, *are defined below where* $\mathbf{x} \in \text{AADD}$, $n \in \text{ID}$, $fs \in \mathcal{P}(\text{ID})$ *and* $ts \in \mathcal{P}(\mathbb{S} \times \text{ID})$*:*

$$s ::= \varnothing_s \mid c \mid s_1.s_2 \quad t ::= \#\text{text}_n[s]_{fs} \quad \mathbf{e} ::= s_n[\mathbf{a}, \mathbf{f}]^{ts}_{fs} \quad \mathbf{a} ::= \varnothing_a \mid \mathbf{x} \mid s_n[\mathbf{tf}]_{fs} \mid \mathbf{a}_1 \odot \mathbf{a}_2$$

$$\mathbf{doc} ::= \#\text{doc}_d[\varnothing_e]^{ts}_{fs} \,\&\, \mathbf{g} \mid \#\text{doc}_d[\mathbf{e}]^{ts}_{fs} \,\&\, \mathbf{g} \mid \#\text{doc}_d[\mathbf{x}]^{ts}_{fs} \,\&\, \mathbf{g}$$

$$\mathbf{f} ::= \varnothing_f \mid \mathbf{x} \mid t \mid \mathbf{e} \mid \mathbf{f}_1 \otimes \mathbf{f}_2 \quad \mathbf{g} ::= \varnothing_g \mid \mathbf{x} \mid t \mid \mathbf{e} \mid s_n[\mathbf{tf}]_{fs} \mid \mathbf{g}_1 \oplus \mathbf{g}_2 \quad \mathbf{tf} ::= \varnothing_{tf} \mid \mathbf{x} \mid t \mid \mathbf{tf}_1 \oslash \mathbf{tf}_2$$

where the operations $.$, $\oslash$, $\otimes$, $\odot$ *and* $\oplus$ *are associative with identities* $\varnothing_s$, $\varnothing_{tf}$, $\varnothing_f$, $\varnothing_a$ *and* $\varnothing_g$, *respectively; the* $\odot$ *and* $\oplus$ *operations are commutative; and all data are equal up to the properties of* $.$, $\oslash$, $\otimes$, $\odot$ *and* $\oplus$. *Data does not contain*

repeated identifiers and abstract addresses; element nodes contain attributes with distinct names. The set of DOM data *is* $\mathbf{d} \in \textsc{Data} \triangleq \mathbb{E} \cup \mathbb{F} \cup \mathbb{TF} \cup \mathbb{A} \cup \mathbb{G} \cup \mathbb{D}$.

When the type of data is clear from the context, we drop the subscripts for empty data and write e.g. $\varnothing$ for $\varnothing_f$. We drop the forest and tag listeners when not relevant to the discussion and write e.g. $s_n[\mathbf{a}, \mathbf{f}]$ for $s_n[\mathbf{a}, \mathbf{f}]^{ts}_{fs}$. Given the set of DOM data Data, there is an associated address function, $\mathsf{Adds}(.)$, which returns the set of context holes present in the data. Context application $\mathbf{d}_1 \circ_{\mathbf{x}} \mathbf{d}_2$ denotes the standard substitution of $\mathbf{d}_2$ for $\mathbf{x}$ in $\mathbf{d}_1$ ($\mathbf{d}_1[\mathbf{d}_2/\mathbf{x}]$) provided that $\mathbf{x} \in \mathsf{Adds}(\mathbf{d}_1)$ and the result is well-typed, and is otherwise undefined.

DOM heaps A DOM heap is a mapping from addresses, $x \in \textsc{Addr} \triangleq \textsc{AAdd} \uplus \{\mathcal{D}\}$, to DOM data. DOM heaps are subject to structural invariants to ensure that they are well-formed. In particular, a context hole $\mathbf{x}$ must not be reachable from the abstract address $\mathbf{x}$ in the domain of the heap. For instance, $\{\mathbf{x} \mapsto s_n[\varnothing, \mathbf{y}], \mathbf{y} \mapsto s'_m[\varnothing, \mathbf{x}]\}$ is not a DOM heap due to the cycle. We capture this by the reachability relation $\rightsquigarrow$ defined as: $x \rightsquigarrow \mathbf{y} \iff \mathbf{y} \in \mathsf{Adds}(\mathbf{h}(x))$, for heap $\mathbf{h}$ and address $x \in \textsc{Addr}$. We write $\rightsquigarrow^+$ to denote the transitive closure of $\rightsquigarrow$.

Definition 2. *The set of* DOM heaps *is:* $\mathbf{h} \in \textsc{DOMHeap} \subseteq (\{\mathcal{D}\} \rightharpoonup \mathbb{D}) \cup (\textsc{AAdd} \overset{fin}{\rightharpoonup} \textsc{Data})$ *provided that for all* $\mathbf{h} \in \textsc{DOMHeap}$ *and* $x \in \textsc{Addr}$ *the following hold:*

1. *identifiers and context holes are unique across* $\mathbf{h}$*;*
2. $\neg \exists \mathbf{x}.\ \mathbf{x} \rightsquigarrow^+ \mathbf{x}$*;*
3. *context holes in* $\mathbf{h}$ *are associated with data of correct type:*
$$\forall x, \mathbf{y}.\ \mathbf{y} \in \mathsf{Adds}(\mathbf{h}(x)) \wedge \mathbf{y} \in \mathrm{dom}(\mathbf{h}) \Rightarrow \exists \mathbf{d}.\ \mathbf{h}(x) \circ_{\mathbf{y}} \mathbf{h}(\mathbf{y}) = \mathbf{d}$$

DOM Heap composition, $\bullet : \textsc{DOMHeap} \times \textsc{DOMHeap} \rightharpoonup \textsc{DOMHeap}$*, is the standard disjoint function union provided that the resulting heap meets the constraints above. The* empty DOM heap*,* $\mathbf{0}$*, is a function with an empty domain.*

Definition 3. *The* abstract (de)allocation relation*,* $\approx : \textsc{DOMHeap} \times \textsc{DOMHeap}$*, is defined as follows where* $*$ *denotes the reflexive transitive closure of the set.*

$$\approx \triangleq \left\{ (\mathbf{h}_1, \mathbf{h}_2), (\mathbf{h}_2, \mathbf{h}_1) \,\middle|\, \exists x, \mathbf{d}_1, \mathbf{d}_2, \mathbf{x}.\ \mathbf{h}_1(x) = (\mathbf{d}_1 \circ_{\mathbf{x}} \mathbf{d}_2) \wedge \mathbf{h}_2 = \mathbf{h}_1[x \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \right\}^*$$

During abstract allocation (from $\mathbf{h}_1$ to $\mathbf{h}_2$), part of the data $\mathbf{d}_2$ at address x is split and promoted to a fresh abstract address $\mathbf{x}$ in the heap leaving the context hole $\mathbf{x}$ behind in its place. Dually, during abstract deallocation (from $\mathbf{h}_2$ to $\mathbf{h}_1$) the context hole $\mathbf{x}$ in DOM data $\mathbf{d}_1$ is replaced by its associated data $\mathbf{d}_2$ at abstract address $\mathbf{x}$, removing $\mathbf{x}$ from the domain of the heap in doing so.

3.2 DOM Assertions

DOM assertions comprise heap assertions describing DOM heaps such as those in Fig. 1. DOM heap assertions are defined via DOM data assertions describing the underlying DOM structure such as nodes, forests and so forth. As we show later, pure assertions such as $\mathsf{out}(\text{A}, \text{S})$ in §2 are derived assertions defined in Fig. 4.

Definition 4. *The* DOM assertions, $\psi \in \text{DOMAsst}$, *and* DOM data assertions, $\phi \in \text{DOMDAsst}$, *are defined as follows where* $\alpha, \text{A}, \text{N}, \cdots$ *denote logical variables.*

$$
\begin{array}{rll}
\psi ::= \mathcal{D} \mapsto \phi \mid \alpha \mapsto \phi & & \text{DOM heap assertions} \\
\phi ::= \text{false} \mid \phi_1 \Rightarrow \phi_2 \mid \exists \text{X}.\ \phi \mid \text{V} \mid \alpha \mid \phi_1 \circ_\alpha \phi_2 \mid \Diamond \alpha & & \text{classical} \mid \text{context hole} \\
\quad \mid \#\text{text}_{\text{N}}[\phi]_{\text{F}} \mid \text{s}_{\text{N}}[\phi_1,\phi_2]^{\text{E}}_{\text{F}} \mid \text{s}_{\text{N}}[\phi]_{\text{F}} \mid \#\text{doc}_{\text{N}}[\phi_1]^{\text{E}}_{\text{F}} \,\&\, \phi_2 \mid \varnothing_e & & \text{nodes} \mid \text{empty doc. element} \\
\quad \mid \varnothing_s \mid \phi_1.\phi_2 \mid \varnothing_a \mid \phi_1 \odot \phi_2 \mid \varnothing_f \mid \phi_1 \otimes \phi_2 & & \text{strings} \mid \text{attr. sets} \mid \text{forests} \\
\quad \mid \varnothing_g \mid \phi_1 \oplus \phi_2 \mid \varnothing_{tf} \mid \phi_1 \oslash \phi_2 & & \text{groves} \mid \text{text forests}
\end{array}
$$

The $\mathcal{D} \mapsto \phi$ assertion describes a single-cell DOM heap at document address $\mathcal{D}$; similarly, the $\alpha \mapsto \phi$ describes a single-cell DOM heap at the abstract address denoted by α. For data assertions, classical assertions are standard. The V is a logical variable describing DOM data. The α is a logical variable denoting a context hole; the $\phi_1 \circ_\alpha \phi_2$ describes data that is the result of replacing the context hole α in ϕ_1 with ϕ_2; $\Diamond\alpha$ describes data that contains the context hole α. The node assertions respectively describe element, text, attribute and document nodes with their data captured by the corresponding sub-assertions. The $\varnothing_e$, $\varnothing_s$, $\varnothing_a$, $\varnothing_f$, $\varnothing_g$ and $\varnothing_{tf}$ describe an empty document element, string, attribute set, forest, grove and text forest, respectively. Similarly, $\phi_1.\phi_2$, $\phi_1 \odot \phi_2$, $\phi_1 \otimes \phi_2$, $\phi_1 \oplus \phi_2$ and $\phi_1 \oslash \phi_2$ respectively describe a string, attribute set, forest, grove and text forest that can be split into two, each satisfying the corresponding sub-assertion.

3.3 PLDOMLogic

We show how to reason about client programs that call the DOM. Our DOM specification is agnostic to the client programming language and we can reason about programs in any language with an SL-based program logic. To this end, given an arbitrary programming language, PL, with an SL-based program logic, PLLogic, we show how to extend PLLogic to PLDOMLogic, in order to enable DOM reasoning. Later in §4, we present a particular instance of PLDOMLogic for JavaScript, and use it to reason about JavaScript clients that call the DOM.

States We assume the underlying program states of PLLogic to be modelled as elements of a *PCM* (partial commutative monoid) $(\text{PLStates}, \circ, 0_{\text{PL}})$, where $\circ$ denotes state composition, and 0_{PL} denotes the unit set. To reason about the DOM operations, in PLDOMLogic we extend the states of PLLogic to incorporate DOM heaps; that is, we define a program state to be a pair, $(h, \mathbf{h})$, comprising a PL state $h \in \text{PLStates}$, and a DOM heap $\mathbf{h} \in \text{DOMHeaps}$.

Definition 5. *Given the PCM of* PL, *the set of* PLDOMLogic program states *is* $\Sigma \in \text{State} \triangleq \text{PLStates} \times \text{DOMHeap}$. State composition, $+ : \text{State} \times \text{State} \rightharpoonup \text{State}$, *is defined component-wise as* $+ \triangleq (\circ, \bullet)$ *and is not defined if composition on either component is undefined. The* unit set *is* $I \triangleq \{(h, \mathbf{0}) \mid h \in 0_{\text{PL}}\}$.

Assertions We assume the PLLogic assertions to include: i) standard classical assertions; ii) standard boolean assertions; iii) standard SL assertions; and iv) an assertion to describe the PL variable store as seen in §2 of the form $\mathsf{store}(\dots)$. In PLDOMLogic we extend the PLLogic assertions with those of DOM (Def. 4), *semantic implication* $\Rrightarrow$, and the *semantic magic wand* $\rightsquigarrow\!\!*$, described shortly.

Definition 6. *The set of* PLDOMLogic assertions, $P \in \textsc{Asst}$, *is defined as follows in the binding order* $*, \Rightarrow, \Rrightarrow, -\!\!*, \sim\!\!*$, *with* $\ominus \in \{\in, =, <, \leq, \subset, \subseteq\}$*:*

$$
\begin{array}{rll}
P, Q ::= & \text{false} \mid P \Rightarrow Q \mid \exists \mathrm{x}.\ P \mid E_1 \ominus E_2 & \text{Classical} \mid \text{Boolean assertions} \\
& \mid \text{emp} \mid P * Q \mid P -\!\!* Q & \text{SL assertions} \\
& \mid \mathsf{store}(\overline{\mathtt{x}_i : \mathrm{v}_i}) \mid \Lambda & \text{variable store} \mid \text{PLLogic-specific assertions} \\
& \mid \psi \mid P \Rrightarrow Q \mid P \sim\!\!* Q & \text{DOM} \mid \text{Structural assertions}
\end{array}
$$

Assertions are interpreted as sets of program states (Def. 5). Classical and boolean assertions are standard. The emp assertion describes an empty program state in the unit set I; the $P * Q$ describes a state that can be split into two substates satisfying P and Q. The $-\!\!*$ connective is the right adjunct of $*$, i.e. $P * (P -\!\!* Q) \Rightarrow Q$. Informally, a state that satisfies $P -\!\!* Q$ is one that is *missing* P, and when combined with P, it satisfies Q. The $\mathsf{store}(\overline{\mathtt{x}_i : \mathrm{v}_i})$ describes a variable store in PL where variables $\overline{\mathtt{x}_i}$ have values $\overline{\mathrm{v}_i}$, respectively. The Λ describes states of the form $(h, \mathbf{0})$ where h satisfies Λ. Dually, the ψ describes states of the form $(h, \mathbf{h})$ where $h \in 0_{\text{PL}}$ and $\mathbf{h}$ satisfies ψ. The $P \Rrightarrow Q$ assertion denotes *semantic implication* and integrates logical implication ($\Rightarrow$) with abstract (de)allocation on DOM heaps (Def. 3). The $\sim\!\!*$ connective is the *semantic right adjunct* of $*$: $P * (P \sim\!\!* Q) \Rrightarrow Q$. It is similar to $-\!\!*$ and incorporates the $\approx$ relation on DOM heaps. Intuitively, a state that satisfies $P \sim\!\!* Q$ is one that is missing P, such that when combined with P *and* undergone a number of (possibly zero) abstract (de)allocations, it satisfies Q. We write $E_1 \dot{\ominus} E_2$ for $E_1 \ominus E_2 \wedge \text{emp}$.

Programming language, proof rules and soundness We extend the programming language of PL with the operations of our DOM fragment (e.g. `getAttribute` in §2.1). The proof rules of PLDOMLogic are those of PLLogic with the exception of the rule of consequence: we generalise the premise to allow semantic implication ($\Rrightarrow$) between assertions rather than logical implication ($\Rightarrow$). We further extend the proof rules with the axioms of DOM operations, DOMAx, defined shortly in §3.4 below. The modified rule of consequence and the rule for DOM axioms are given below. We prove PLDOMLogic sound in [17].

$$
\frac{P \Rrightarrow P' \quad \{P'\}\ \mathtt{C}\ \{Q'\} \quad Q' \Rrightarrow Q}{\{P\}\ \mathtt{C}\ \{Q\}}\ (\text{Con}) \qquad \frac{(P, \mathtt{C}, Q) \in \text{DOMAx}}{\{P\}\ \mathtt{C}\ \{Q\}}\ (\text{Ax})
$$

3.4 DOM Operations and Axioms

We formally axiomatise the behaviour of a DOM operations associated with our fragment. In Fig. 3 we give a select number of axioms including those of the operations used in the examples of this paper. The behaviour of some of the operations is captured by several axioms; we have omitted analogous cases. A full list of DOM operations modelled and their axioms, DOMAx, are given in [17].

The assertions in the pre- and postconditions of axioms are of the form $\mathsf{store}(\cdots) * \psi$ where the store predicate states the value associated with each program variable, and ψ is a DOM assertion that describes the operation footprint. Since the DOM library may be called by different client programs written in

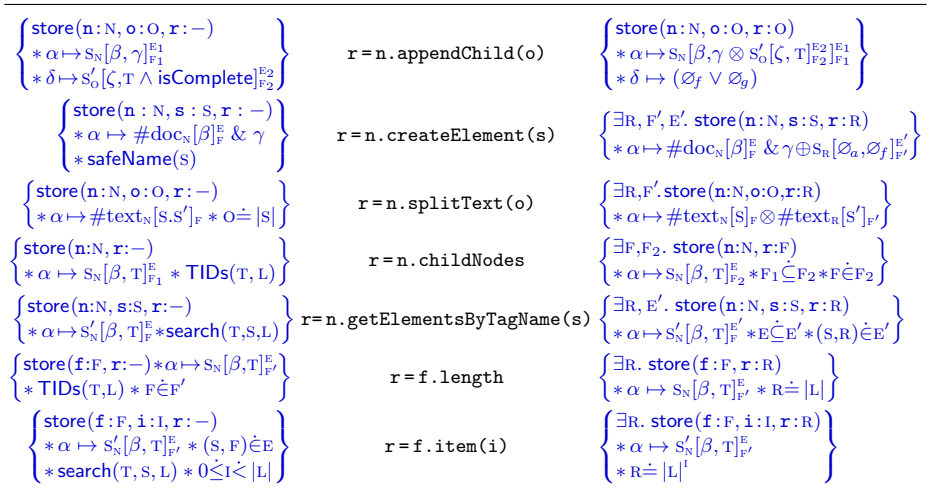

Fig. 3: DOM Core Level 1 axioms (excerpt)

different programming languages, store denotes a *black-box* predicate that can be instantiated to describe a variable store in the client programming language. In §4 we reason about JavaScript client programs that call the DOM and thus instantiate store to describe the JavaScript variable store emulated in the heap.

We now describe of the DOM operations in Fig. 3 and their axioms, delaying the description of the last four operations until §3.5.

`n.appendChild(o)`: when `n` and `o` both identify nodes, this operation appends `o` to the end of `n`'s child list and returns `o`. It fails if `o` is an ancestor of `n` (otherwise it would introduce a cycle and break the DOM structure); or if `n` is a text node or a document node with a non-empty document element; or if `o` is an attribute or a document node. Fig. 3 shows the axiom for when `o` is an element node (O). To ensure that O is not an ancestor of `n`, we require the entire subtree at `o` to be *separate* from the subtree at `n`. This is achieved by the isComplete assertion and the separating conjunction $*$. The isComplete is a derived assertion defined in Fig. 4. It describes DOM data with no context holes. The postcondition leaves $\varnothing_f \vee \varnothing_g$ in place of `o` once moved since we do not know if `o` has come from a forest or grove position. The disjunction leaves the choice to the frame.

`n.createElement(s)`: when `n` identifies a document node, it creates a new element named `s`, and returns its identifier. The new element has no attributes or children and resides in the grove. The grove in the precondition is thus extended with the new node in the postcondition. The safeName(s) assertion is defined in Fig. 4 ensures that the tag name does not contain the invalid character '#'.

`n.splitText(o)`: when `n` identifies a text node and `o` denotes an integer, it breaks the data of `n` into two text nodes at offset `o` (indexed from 0), keeping both nodes in the tree as siblings. It fails when `o` is an invalid offset (i.e. negative or greater than the length). The return value is the identifier of the new node.

$$\mathsf{isComplete} \triangleq \neg\exists\alpha.\ \Diamond\alpha \qquad \mathsf{safeName}(\mathrm{S}) \triangleq \neg\exists \mathrm{S}_1, \mathrm{S}_2.\ \mathrm{S} \doteq \mathrm{S}_1.\text{`\#'}.\mathrm{S}_2$$

$$\mathsf{val}(\mathrm{T}, \mathrm{S}) \triangleq (\mathrm{T} \doteq \varnothing_{tf} * \mathrm{S} \doteq \text{``''}) \vee (\exists \mathrm{N}, \mathrm{S}_1, \mathrm{S}_2, \mathrm{T}'.\ \mathrm{T} \doteq \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1]_{\_} \oslash \mathrm{T}' * \mathsf{val}(\mathrm{T}', \mathrm{S}_2) * \mathrm{S} \doteq \mathrm{S}_1.\mathrm{S}_2)$$

$$\mathsf{out}(\mathrm{A}, \mathrm{S}) \triangleq (\mathrm{A} \doteq \varnothing_a) \vee (\exists \mathrm{S}', \mathrm{N}, \mathrm{T}, \mathrm{A}'.\ \mathrm{A} \doteq \mathrm{S}'_{\mathrm{N}}[\mathrm{T}]_{\_} \odot \mathrm{A}' * \mathrm{S} \neq \mathrm{S}' * \mathsf{out}(\mathrm{A}', \mathrm{S}))$$

$$\mathsf{TIDs}(\mathrm{T}, \mathrm{L}) \triangleq (\mathrm{L} \doteq [\,] * \mathrm{T} \doteq \varnothing_f) \vee \big(\exists \mathrm{N}, \mathrm{S}, \mathrm{A}, \mathrm{F}, \mathrm{T}', \mathrm{L}'.\ \mathrm{L} \doteq \mathrm{N}{:}\mathrm{L}' * (\mathrm{T} \doteq \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\_} \otimes \mathrm{T}' \vee \mathrm{T} \doteq \mathrm{S}_{\mathrm{N}}[\mathrm{A},\mathrm{F}]^{\_} \otimes \mathrm{T}') * \mathsf{TIDs}(\mathrm{T}', \mathrm{L}')\big)$$

$$\mathsf{search}(\mathrm{T},\mathrm{S},\mathrm{L}) \triangleq (\mathrm{T} \doteq \varnothing_f * \mathrm{L} \doteq [\,]) \vee (\exists \mathrm{N}, \mathrm{S}', \mathrm{T}'.\ \mathrm{T} \doteq \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\_} \otimes \mathrm{T}' * \mathsf{search}(\mathrm{T}', \mathrm{S}, \mathrm{L}))$$
$$\vee \big(\exists \mathrm{S}', \mathrm{N}, \mathrm{T}_1, \mathrm{T}_2, \mathrm{L}_1, \mathrm{L}_2.\ \mathrm{T} \doteq \mathrm{S}'_{\mathrm{N}}[-, \mathrm{T}_1]^{\_} \otimes \mathrm{T}_2 * \mathsf{search}(\mathrm{T}_1, \mathrm{S}, \mathrm{L}_1) * \mathsf{search}(\mathrm{T}_2, \mathrm{S}, \mathrm{L}_2)$$
$$* (\mathrm{S} \doteq \mathrm{S}' \vee \mathrm{S} \doteq \text{``*''} \Rightarrow \mathrm{L} \doteq \mathrm{N}{:}(\mathrm{L}_1 +\!\!+ \mathrm{L}_2)) * (\mathrm{S} \neq \mathrm{S}' \wedge \mathrm{S} \neq \text{``*''} \Rightarrow \mathrm{L} \doteq \mathrm{L}_1 +\!\!+ \mathrm{L}_2)\big)$$

Fig. 4: Derived DOM assertions

Our specifications have smaller footprints than those of [9,22]. In particular, the axiom of `appendChild` requires a substantial *overapproximation* of the footprint due to the reasons discussed in §2.1, namely the need for a *linking context* (see page 7). This axiom is given below using MCL [5] (adapted to our notation):

$$\big\{(C \bullet_\alpha \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]) \bullet_\beta \mathrm{S}'_{\mathrm{O}}[\mathrm{A}', \mathrm{T}]\big\}\ \texttt{n.appendChild(o)}\ \big\{(C \bullet_\alpha \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma \otimes \mathrm{S}'_{\mathrm{O}}[\mathrm{A}', \mathrm{T}]]) \bullet_\beta \varnothing_f\big\}$$

This axiom is not small enough: the only parts required by `appendChild` are the tree at O being moved, and the element N whose children are extended by O. However, as before the precondition above also requires the linking context C.

3.5 Live Collections

The DOM API provides several interfaces for traversing DOM trees based on *live collections* of nodes, such as the *NodeList* interface in DOM CL1-4. DOM CL 4 also introduces the *HTMLCollection* interface for live collections of element nodes. We describe our model of live collections in terms of NodeLists. However, our model is abstract and captures the behaviour of both NodeLists and HTMLCollections.

The *NodeList* interface is an ordered collection of nodes. NodeLists are *live* in that they dynamically reflect document changes. Several DOM operations return NodeLists. For example, `n.getElementsByTagName(s)` returns a NodeList (using depth-first, left-to-right search) containing the identifiers of the elements named `s` underneath the tree rooted at `n`. Given the DOM tree of Fig. 1a, when `n`=4 and `s`="img", then `r = n.getElementsByTagName(s)` yields `r`=[3, 8, 2]. However, since NodeLists are live, if node 8 is later removed from the document, then `r`=[3, 2]. When `s`="*" denoting a wildcard, then the resulting NodeList must contain the identifiers of *all* element nodes underneath `n`. For instance, with the DOM tree of Fig. 1a, when `n`=4 and `s`="*", then `r = n.getElementsByTagName(s)` yields `r`=[9, 3, 8, 6, 2]. This operation may be called on both document and element nodes. We thus associate each such node with a *set of tag listeners*, *ts*. Each listener is of the form (s, fid) where s denotes the search string (e.g. "img" in the example above) and $\mathit{fid} \in \mathrm{ID}$ denotes the identifier of the resulting NodeList.

The `n.childNodes` operation also returns a NodeList, containing the identifiers of the immediate children of `n`. For instance, with the DOM tree of Fig. 1a,

when n=4, then r = n.childNodes returns r=[9, 6]. Again, the value of r is live and dynamically reflects the changes to the child forest of n. The n.childNodes operation may be called on *any* DOM node. We therefore associate each DOM node with a *set of forest listeners*, *fs*. Each forest listener, *fid* ∈ ID, denotes the identifier of a NodeList. Our specification is the first that faithfully models the behaviour of NodeLists. In particular, both [9] and [22] associate a single forest listener with DOM nodes and consequently admit behaviours that are not guaranteed by the standard. We proceed with the NodeList axioms in Fig. 3.

n.childNodes: when n=N, this operation returns (the identifier of) a forest listener NodeList F associated with N. Fig. 3 shows the axiom for when N is an element. When asked for a forest listener NodeList, a node may either return an existing one, or generate a fresh one and extend its set with it. This flexibility is due to an under-specification in the standard. Thus, in the postcondition the original set F_1 is extended to F_2 ($F_1 \dot{\subseteq} F_2$) with return value $F \in F_2$. The TIDs(T,L) assertion is defined in Fig. 4 and states that list L contains the top-level node identifiers (from left to right) of the forest denoted by T. For instance, TIDs(T, [9, 6]) holds in Fig. 1a when T denotes the child forest of node 4 (named "body"). As such, the TIDs(T,L) in the precondition stipulates that T contain enough resource for compiling a list of the immediate children of N (i.e. the top-level nodes in T).

n.getElementsByTagName(s): when n=N and s=S, this operation returns (the identifier of) a NodeList containing the identifiers of the elements with tag name S in the forest underneath N. The axiom in Fig. 3 describes the case when N is an element node. The original set of tag listeners E is extended to E′ with (S, R) ∈ E′ where R is the return value. The search(T, S, L) assertion is defined in Fig. 4 and describes the search result of getElementsByTagName (i.e. the list L contains the identifiers of those element nodes in the forest T whose name matches S). For instance, when T denotes the child forest of node 4 (named "body") in Fig. 1a, then both search(T, "img", [3, 8, 2]) and search(T, "*", [9, 3, 8, 6, 2]) hold. As such, the search(T, S, L) in the precondition ensures that T contains enough resource for compiling a list of elements named S.

f.length: when f=F identifies a NodeList, its length is returned. The axiom in Fig. 3 describes the case when F is a forest listener NodeList on element N; the return value is the number of N's immediate children. This is captured by TIDs(T,L) stipulating that list L contains the identifiers of those nodes at the top level of child forest T. The return value is thus the length of L (i.e. |L|).

f.item(i): this is analogous to f.length with $|L|^I$ denoting the Ith item of L. The axiom in Fig. 3 describes the case when F is a tag listener NodeList on N.

4 Verifying JavaScript Programs that Call the DOM

We instantiate the method described in §3.3 to extend the SL-based JavaScript program logic (hereafter JSLogic) in [7], to JSDOMLogic, in order to enable DOM reasoning. We then use JSDOMLogic to reason about a realistic ad blocker program in §4.1, and a further ad blocker in [17]. These examples are interesting as they combine JavaScript heap reasoning with DOM reasoning.

JSLogic States The states of JSLogic are *JavaScript heaps*. A JavaScript heap, $h \in \text{JSHEAP}$, is a partial function mapping *references*, which are pairs of memory locations and field names, to values. A heap cell is written $(l, x) \mapsto 7$, stating that the object at l has a field named x and holds value 7. An empty JavaScript heap is denoted by 0_{JS}; JavaScript heap composition, $\circ : \text{JSHEAP} \times \text{JSHEAP} \rightharpoonup \text{JSHEAP}$, is the standard disjoint function union. The PCM of JavaScript heaps is $(\text{JSHEAP}, \circ, \{0_{\text{JS}}\})$. The states of JSDOMLogic are then pairs of the form $(h, \mathbf{h})$, comprising a JavaScript heap h, and a DOM heap $\mathbf{h}$ (see Def. 5).

JSLogic Assertions, programming language and proof rules As stipulated by Def. 6, the JSLogic assertions include the standard boolean, classical and SL assertions. JSLogic further includes JavaScript heap assertions of the form $(E_1, E_2) \mapsto E_3$, describing a single-cell JavaScript heap. The variable store in JavaScript is emulated in the heap. As required by Def. 6, JSLogic introduces a derived assertion $\mathsf{store}(\overline{\mathtt{x}_i : \mathrm{V}_i})$, describing the JavaScript variable store in the heap where variables $\overline{\mathtt{x}_i}$ have values $\overline{\mathrm{V}_i}$. The programming language of JSLogic is a broad subset of the JavaScript language [7]. The JSLogic assertions, their semantics, the definition of store, and the JSLogic proof rules are given in [7].

4.1 A JavaScript Ad Blocker

We use JSDOMLogic to reason about an *ad blocker* script used for blocking the images from untrusted sources in a DOM tree. The `adBlocker1(n)` program in Fig. 5 compiles a NodeList containing all "img" elements in the tree rooted at `n` by calling the `getElementsByTagName` operation. It then iterates over this NodeList, sanitising each image by executing the `sanitiseImg` program in §2.

At each iteration I, the subtree at node n=N is described by $\mathsf{tree}(\mathrm{I}, \mathrm{E})$ where T_I denotes the child forest of N at iteration I, and E denotes the tag listener set associated with N, and L denotes the list of "img" elements below N.[3]

Since we iterate over the "img" elements in L and inspect their attributes, we need to *partition* them in into three categories: i) *empty*: without a "src" attribute; ii) *untrusted*: with a"src" attribute and a blacklisted value; iii) *trusted*: with a "src" attribute and a trusted value. At each iteration, if the node considered is untrusted, it is sanitised and removed from the untrusted category. We thus define a fourth category, *sanitised*, including those elements whose values were initially blacklisted and are later sanitised. This is captured by $\mathsf{partition}(\mathrm{I})^3$. The first part states that the list of "img" elements L can be partitioned into the three categories described above where $\mathrm{L} \equiv \mathrm{S}$ states that set S is a permutation of list L. The second part states that list L has been processed up to index I; i.e. the sanitised category S_s includes all the untrusted elements in L up to index I. The last four parts describe the "img" elements according to their category.

[3] All free logical variables on the right-hand side are parameters of the predicate on the left. We omit them for readability as they do not change throughout the execution. By contrast, the iteration number I, and the tag listeners E of node N may change (the latter may grow by `getElementsByTagName`) and are explicitly parameterised.

$\mathsf{cache}(\mathrm{C}) \triangleq \circledast_{\mathrm{F}\in\mathcal{X}} \big((\mathrm{C},\mathrm{F})\mapsto 1 \vee (\mathrm{C},\mathrm{F})\mapsto 0\big)$ $\quad \mathsf{unfld}(\mathrm{I},\mathrm{E}) \triangleq \exists \overline{\alpha,\beta,\gamma}^{\mathrm{L}}.\, \mathsf{partition}(\mathrm{I}) * (\forall \mathrm{I}.\, \mathsf{partition}(\mathrm{I}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I},\mathrm{E}))$

$\mathsf{tree}(\mathrm{I},\mathrm{E}) \triangleq \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A},\mathrm{T}_{\mathrm{I}}]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{search}(\mathrm{T}_{\mathrm{I}},\text{“img”},\mathrm{L})$ $\quad \mathsf{fld}(\mathrm{I},\mathrm{E}) \triangleq \mathsf{tree}(\mathrm{I},\mathrm{E}) * \big((\forall \mathrm{I},\mathrm{E}.\, \mathsf{tree}(\mathrm{I},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{E})) \wedge \mathrm{emp}\big)$

$\mathsf{rem}(\mathrm{I}) \triangleq \exists \mathrm{S}_s.\ \mathrm{S}_s \doteq \mathrm{S}_u \cap \{|\mathrm{L}|^{\mathrm{J}} \mid \mathrm{J} < \mathrm{I}\} * \beta \mapsto \varnothing_g \bigoplus_{\mathrm{J}\in\mathrm{S}_s} \mathrm{A}_{\mathrm{J}}$

$\mathsf{partition}(\mathrm{I}) \triangleq \mathrm{L} \doteq \mathrm{S}_e \uplus \mathrm{S}_u \uplus \mathrm{S}_t * \exists \mathrm{S}_s.\ \mathrm{S}_s \doteq \mathrm{S}_u \cap \{|\mathrm{L}|^{\mathrm{J}} \mid \mathrm{J} < \mathrm{I}\} * \circledast_{\mathrm{J}\in\mathrm{S}_s} \big(\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\beta_{\mathrm{J}} \odot \mathrm{src}_{\mathrm{M}_{\mathrm{J}}}[\#\mathrm{text}_{-}[\mathrm{S}]_{-}]_{\mathrm{F}'_{\mathrm{J}}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}}\big)$
$\circledast_{\mathrm{J}\in\mathrm{S}_e} (\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\mathrm{A}_{\mathrm{J}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}} * \mathsf{out}(\mathrm{A}_{\mathrm{J}},\mathrm{src})) * \circledast_{\mathrm{J}\in\mathrm{S}_t} \big(\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\beta_{\mathrm{J}} \odot \mathrm{src}_{\mathrm{M}_{\mathrm{J}}}[\mathrm{A}_{\mathrm{J}}]_{\mathrm{F}'_{\mathrm{J}}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}} * \mathsf{val}(\mathrm{A}_{\mathrm{J}},\mathrm{V}_{\mathrm{J}}) * \neg\mathsf{isB}(\mathrm{V}_{\mathrm{J}})\big)$
$\circledast_{\mathrm{J}\in\mathrm{S}_u\setminus\mathrm{S}_s} \big(\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\beta_{\mathrm{J}} \odot \mathrm{src}_{\mathrm{M}_{\mathrm{J}}}[\mathrm{A}_{\mathrm{J}}]_{\mathrm{F}'_{\mathrm{J}}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}} * \mathsf{val}(\mathrm{A}_{\mathrm{J}},\mathrm{V}_{\mathrm{J}}) * \mathsf{isB}(\mathrm{V}_{\mathrm{J}})\big)$

$\{\mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}-,\mathtt{len}{:}-,\mathtt{i}{:}-,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{fld}(0,\mathrm{E}) * \mathsf{rem}(0)\}$

1. `adBlocker1(n)` $\triangleq$ {
2. `  imgs = n.getElementsByTagName("img");`
   $\{\exists \mathrm{R},\mathrm{E}'.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}-,\mathtt{i}{:}-,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(0) * \mathsf{fld}(0,\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}'\}$
3. `  len = imgs.length; i = 0;`
   $\{\exists \mathrm{R},\mathrm{E}'.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}0,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(0) * \mathsf{fld}(0,\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}'\}$
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathsf{fld}(\mathrm{I},\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}|\}$
4. `  while(i<len){`
5. `    c = imgs.item(i);`
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathsf{fld}(\mathrm{I},\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}|\}$
   //Apply derivation steps in (10)-(12).
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}| * \mathsf{unfld}(\mathrm{I},\mathrm{E}') * ((\forall \mathrm{I},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{E})) \wedge \mathrm{emp})\}$
6. `    sanitiseImg(c,cat);`
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}{+}1) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}| * \mathsf{unfld}(\mathrm{I}{+}1,\mathrm{E}') * ((\forall \mathrm{I},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{E})) \wedge \mathrm{emp})\}$
   // Apply derivation steps in (12)-(14).
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}{+}1) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}| * \mathsf{fld}(\mathrm{I}{+}1, \mathrm{E}')\}$
7. `    i = i+1;`
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}| * \mathsf{fld}(\mathrm{I}, \mathrm{E}')\}$
8. } } $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}-,\mathtt{len}{:}-,\mathtt{i}{:}-,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(|\mathrm{L}|) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}| * \mathsf{fld}(|\mathrm{L}|, \mathrm{E}')\}$

Fig. 5: A proof sketch of the `adBlocker1` program[3]

The partition predicate describes the “img” elements in L only and does not include the *remainder* of the subtree at N. At every iteration, this remainder is untouched and the modified parts are in the partitions. We thus describe the remainder for an arbitrary iteration I as $\forall \mathrm{I}.\ \mathsf{partition}(\mathrm{I}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I},\mathrm{E})$, i.e. the entire tree for that iteration, $\mathsf{tree}(\mathrm{I},\mathrm{E})$, *minus* its partitions. The *unfolded* tree at iteration I, $\mathsf{unfld}(\mathrm{I},\mathrm{E})$[3], consists of the partitions at I, plus the remainder.

Note that for NodeList operations such as `item` (line 5), we need the *folded* tree ($\mathsf{tree}(\mathrm{I},\mathrm{E})$) with the entire subtree containing the “img” list L, as required by their axioms (Fig. 3). Conversely, for the `sanitiseImg` call (line 6), we need the *unfolded* “img” elements ($\mathsf{partition}(\mathrm{I})$) so that we can access the relevant “img”

node at each iteration. We thus need to move between the folded and unfolded tree depending on the operation considered. The $\mathsf{fld}(\mathrm{I}, \mathrm{E})$ predicate describes the folded tree at iteration I. The first part, $\mathsf{tree}(\mathrm{I}, \mathrm{E})$, describes the resources of the folded tree at iteration I. The second part contains no resources (emp); it simply states that at any iteration I, the folded tree $\mathsf{tree}(\mathrm{I}, \mathrm{E})$, can be *exchanged* for the unfolded tree $\mathsf{unfld}(\mathrm{I}, \mathrm{E})$. As we show in the derivation below, this second part allows us to move from folded to unfolded resources (10-12) and vice versa (12-14), for any I. The bi-implication of (10) follows from the definition of fld and that empty resources (emp) can be freely duplicated. In (11) we eliminate the first universal quantifier. We then eliminate the adjunct $(P * (P \mathrel{-\!\!*} Q) \Rrightarrow Q)$ and arrive at (12). The implication of (13) follows from the definition of unfld and the elimination of the first universal quantifier. To get (14), we eliminate the adjunct, eliminate the existential quantifiers and wrap the definition of fld.

$$
\begin{aligned}
\mathsf{fld}(\mathrm{I}, \mathrm{E}) \Leftrightarrow\ & \mathsf{tree}(\mathrm{I}, \mathrm{E}) * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) \\
& * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (10) \\
\Rightarrow\ & \mathsf{tree}(\mathrm{I}, \mathrm{E}) * (\mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E})) * (\forall \mathrm{I}, \mathrm{E}.\, \mathsf{tree}(\mathrm{I}', \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (11) \\
\Rrightarrow\ & \mathsf{unfld}(\mathrm{I}, \mathrm{E}) * (\forall \mathrm{I}, \mathrm{E}.\, \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (12) \\
\Rightarrow\ & \exists \overline{\alpha, \beta, \gamma}^{\mathrm{L}}.\ \mathsf{partition}(\mathrm{I}) * (\mathsf{partition}(\mathrm{I}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}, \mathrm{E})) \\
& * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (13) \\
\Rrightarrow\ & \mathsf{tree}(\mathrm{I}, \mathrm{E}) * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) \ \Leftrightarrow\ \mathsf{fld}(\mathrm{I}, \mathrm{E}) && (14)
\end{aligned}
$$

Recall that when the value of an attribute node is updated via the `setAttribute` operation, its text forest is replaced with a new text node containing the new value, and its old text forest is added to the grove (see axiom (3)). As such, at each iteration if we sanitise the "src" attribute of `c` (via `sanitiseImg` in line 6), then the old text forest of the "src" attribute is moved to the grove. This is described by the $\mathsf{rem}(\mathrm{I})$ assertion stating that for each attribute node sanitised so far (i.e. those in S_s), the old text forest A_J has been added to the grove.

Recall that `sanitiseImg` (Fig. 2) maintains a local cache of blacklisted URLs, implemented as an object at C with one field per URL (where $(\mathrm{C}, \mathrm{F}) \mapsto 1$ asserts the URL f is blacklisted, and $(\mathrm{C}, \mathrm{F}) \mapsto 0$ asserts that there are no cached results associated with f). We thus define the cache as the collection of all fields (denoted by $\mathcal{X}$) on C with value 1 or 0, where $\circledast$ is the iterated analogue of $*$.

We give a proof sketch of `adBlocker1` in Fig. 5. The precondition consists of the variable store, the cache and the *unprocessed* (iteration 0) tree. The postcondition comprises the store, the cache and the *fully processed* (iteration $|\mathrm{L}|$) tree with the tag listeners of N extended with a new listener for "img".

Concluding remarks We use SSL [25] to formally specify an expressive fragment of DOM Core Level 1, closely following the standard [1]. In comparison to existing work [9,22], our specification i) allows for *local* and *compositional* client specification and verification; ii) can be simply *integrated* with SL-based program logics; and iii) is *faithful* to the standard with respect to the behaviour of live collections. We demonstrate our compositional client reasoning by extending JSLogic [7] to incorporate our DOM specification and verifying functional properties of ad-blocker client programs that call the DOM.

Acknowledgements This research was supported by EPSRC programme grants EP/H008373/1, EP/K008528/1 and EP/K032089/1.

References

1. W3C DOM standard, `www.w3.org/TR/REC-DOM-Level-1/level-one-core.html`.
2. N. Biri and D. Galmiche. A Separation Logic for Resource Distribution. In *FST TCS*, 2003.
3. N. Biri and D. Galmiche. Models and separation logics for resource trees. In *Journal of Logic and Computation*, 2007.
4. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudžiūnienė, A. Schmitt, and G. Smith. A mechanised JavaScript specification. In *POPL*, 2014.
5. C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjunct elimination in context logic for trees. In *Programming Languages and Systems*, 2007.
6. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.
7. P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.
8. P. Gardner, A. Raad, M. Wheelhouse, and A. Wright. Local reasoning for concurrent libraries: mind the gap. In *MFPS*, 2014.
9. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local Hoare Reasoning about DOM. In *PODS*, 2008.
10. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, 1999.
11. S. Jensen, A. Møller, and P.Thiemann. Type analysis for JavaScript. In *SAS*, 2009.
12. S.H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript Web applications. In *ESEC/FSE '11*, 2013.
13. B. S. Lerner, M. Carroll, D. P. Kimmel, H. Q. La Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *WebApps*, 2012.
14. S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.
15. C. Park, S. Won, J. Jin, and S. Ryu. A static analysis of JavaScript web applications in the wild via practical DOM modeling (T). In *ASE*, 2015.
16. M. Parkinson. *Local reasoning for Java.* PhD thesis, Cambridge University, 2006.
17. A. Raad. *(To appear).* PhD thesis, Imperial College, 2016.
18. V. Rajani, A. Bichhawat, D. Garg, Deepak, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *CSF*, 2015.
19. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.
20. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.
21. A. Møller S. H. Jensen, M. Madsen. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *FSE*, 2011.
22. G. Smith. *Local reasoning for web programs.* PhD thesis, Imperial College, 2010.
23. N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *PLDI*, 2013.
24. P. Thiemann. A type safe DOM API. In *DBPL*, 2005.
25. A. Wright. *Structural separation logic.* PhD thesis, Imperial College, 2013.