A Formally Verified Model of Web Components

Achim D. Brucker[1] and Michael Herzberg[2]

[1] Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk
[2] Department of Computer Science, The University of Sheffield, Sheffield, UK
msherzberg1@sheffield.ac.uk

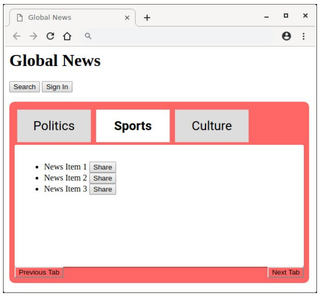
Abstract. The trend towards ever more complex client-side web applications is unstoppable. Compared to traditional software development, client-side web development lacks a well-established component model, i. e., a method for easily and safely reusing already developed functionality. To address this issue, the web community started to adopt *shadow trees* as part of the Document Object Model (DOM). Shadow trees allow developers to "partition" a DOM instance into parts that should be safely separated, e. g., code modifying one part should not unintentionally affect other parts of the DOM.
While shadow trees provide the technical basis for defining web components, the DOM standard neither defines the concept of web components nor specifies the safety properties that web components should guarantee. Consequently, the standard also does not discuss how or even if the methods for modifying the DOM respect component boundaries.
In this paper, we present a formally verified model of web components and define safety properties which ensure that different web components can only interact with each other using well-defined interfaces. Moreover, our verification of the application programming interface (API) of the DOM revealed numerous invariants that implementations of the DOM API need to preserve to ensure the integrity of components.

Keywords: Web component · Shadow tree · DOM · Isabelle/HOL

1 Introduction

The trend towards ever more complex client-side web applications is unstoppable. Compared to traditional software development, client-side web development lacks a well-established component model which allows easily and safely reusing implementations. The Document Object Model (DOM) essentially defines a tree-like data structure (the *node tree*) for representing documents in general and HTML documents in particular.

Shadow trees are a recent addition to the DOM standard [24] to enable web developers to partition the node tree into "sub-trees." The vision of shadow trees is to enable web developers to provide a library of re-usable and customizable widgets. For example, let us consider a multi-tab view called *Fancy Tab*, which is a simplified version of [3].



(a) User view

```
<fancy-tabs>
  <button slot="title">Politics</button>
  <button slot="title" selected>Sports</button>
  <button slot="title">Culture</button>
  <section>content panel 1</section>
  <ul>
    <li>News Item 1 <button>Share</button></li>
    <li>News Item 2 <button>Share</button></li>
    <li>News Item 3 <button>Share</button></li>
  </ul>
  <section>content panel 3</section>
</fancy-tabs>
```

(b) Consumer view

Fig. 1. Our running example: a fancy tab component.

The left-hand side of Fig. 1 shows the rendered output of the widget in use while the right-hand side shows the HTML source code snippet (we will discuss the implementation of Fancy Tab in Sect. 3). It provides a custom HTML tag `<fancy-tabs>` using an HTML template that developers can use to include the widget. Its children will be rendered inside the widget, more precisely, inside its *slots* (elements of type `slot`). It has a slot called "title" and a default slot, which receives all children that do not specify a "slot" attribute.

It is important to understand that slotting does *not change* the structure of the DOM (i. e., the underlying pointer graph): instead, slotting is implemented using special element attributes such as "slot," which control the final rendering. The DOM standard specifies methods that inspect the effect of these attributes such as `assigned_slot`, but the majority of DOM methods do not consider the semantics of these attributes and therefore do not traverse into shadow trees.

This provides an important boundary for client-side code. For example, a JavaScript program coming from the widget developer that changes the style attributes of the "Previous Tab" and "Next Tab" buttons in the lower corners of the widget will not affect buttons belonging to other parts coming from outside, i. e., the application of the widget consumer. Similarly, a JavaScript program that changes the styles of buttons outside of Fancy Tab, such as the navigation buttons, will not have any effect on them, even in the case of duplicate identifiers.

Sadly, the DOM standard neither defines the concept of web components nor specifies the safety properties that they should guarantee, not even informally. Consequently, the standard also does not discuss how or even if the methods for modifying the node tree respect component boundaries. Thus, shadow roots are only the very first step in defining a safe web component model.

Earlier [7], we presented a formalization of the "flat" DOM (called Core DOM) without any support for shadow trees or components. In this paper, we extend the Core DOM with a formal model of shadow trees and slots which we use for defining a *formally verified model of web components* in general and, in particular, the notion of *weak* and *strong component safety*. For all methods that query, modify, or transform the DOM, we formally analyze their level of component safety. In more detail, the contribution of this paper is four-fold:

1. We provide a formal model of web components and their safety guarantees to web developers, enabling a compositional development of web applications,
2. for each method, we formally verify that it is either weakly or strongly component safe, or we provide a proof showing that it is not component safe,
3. we fill the gaps in the standard by explicitly formalizing invariants that are left out in the standard. These invariants are required to ensure that methods in the standard preserve a valid node tree. Finally,
4. we present a formal model of the DOM with shadow roots including the methods for querying, modifying, and transforming DOM instances with shadow roots.

Overall, our work gives web developers the guarantee that their code will respect the component boundaries as long as they abstain from or are careful when using certain DOM methods such as `appendChild` or `ownerDocument`.

On the Relationship Between the "DOM Standard" and the "Shadow DOM Standard." The development of shadow trees started in the context of the dedicated "Shadow DOM" standard [23]. This standard has been considered obsolete since at least March 2018, as shadow trees have been integrated into the DOM standard [24] itself. Hence, we will in the following only refer to the "DOM standard." Our formalization is faithful with respect to the standard in the sense that our formalization passes all relevant compliance test cases provided by the standard authors (the detailed discussion of this matter is out of scope of this paper).

2 Background

In this section, we will introduce the formal background of our work and the formalization of the Core DOM [6, 7], without support for shadow roots.

2.1 Isabelle and Higher-Order Logic

Isabelle/HOL [17] is a generic theorem prover supporting Higher-order Logic (HOL). It supports conservativity checks of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines.

HOL [2, 9] is a classical logic with equality enriched with total polymorphic higher-order functions. HOL is strongly typed, i. e., each expression `e` has a type `'a`, written `e::'a`. In Isabelle, we denote type variables with a prime (e. g., `'a`) instead of Greek letters (e. g., α) that are usually used in textbooks. The type constructor for the function space is written infix: `'a` $\Rightarrow$ `'b`. HOL is centered around the extensional logical equality `_ = _` with type `'a` $\Rightarrow$ `'a` $\Rightarrow$ `bool`, where `bool` is the fundamental logical type. The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type `'a set` can be defined isomorphic to functions of type `'a` $\Rightarrow$ `bool`; the element-of-relation $\_ \in \_$ has the type `'a` $\Rightarrow$ `'a set` $\Rightarrow$ `bool` and corresponds basically to the function application; in contrast, the set comprehension `{_ . _}` (usually written $\{\_ \mid \_\}$ in textbooks) has type `'a set ⇒ ('a ⇒ bool) ⇒ 'a set` and corresponds to the λ-abstraction.

Isabelle/HOL allows defining abstract datatypes. For example, the following statement introduces the option type:

```
datatype 'a option = None | Some 'a
```

Besides the *constructors* `None` and `Some`, there is the match-operation `case x of None ⇒ F | Some a ⇒ G a`, which acts as a case distinction on `x::'a option`. The option type allows us to represent *partial functions* (often called *maps*) as total functions of type `'a ⇒ 'b option`. For this type, we introduce the short-hand `'a ⇀ 'b`. We define `dom f`, called the *domain* of a partial function `f` by the set of all arguments of `f` that do not yield `None`. By restricting the domain of a map to be finite, we can define a type that represents finite maps:

```
typedef ('a, 'b) fmap = {m. finite (dom m)}::('a ⇀ 'b) set
```

In this paper, we will use a short-hand that hides type variables when they are identical to the declaration of a datatype or a type definition. For example, we will write `(_) fmap` instead of `('a, 'b) fmap`. This short-hand notation is provided by [6] as an Isabelle theory, i. e., it is fully supported by Isabelle.

We represent DOM methods directly as HOL functions (*shallow embedding*) using a monad-like syntax that looks similar to Haskell programs. Additionally, we introduce the syntax $\texttt{h} \vdash \texttt{f x} \rightarrow_{\texttt{r}} \texttt{y}$, which is a predicate that evaluates to true if and only if the method `f` invoked on heap `h` and with argument `x` returns the value `y`. Similarly, $\texttt{h} \vdash \texttt{f x} \rightarrow_{\texttt{h}} \texttt{h'}$, refers to the (potentially modified) new heap `h'`. In addition, we introduce syntax for binding the result directly to a variable `y`: $\texttt{y} = |\texttt{h} \vdash \texttt{f x}|_{\texttt{r}}$.

2.2 The Core DOM

Our work is based on the formalization of the Core DOM presented in [6, 7]. The Core DOM describes a tree-like data structure *without* shadow roots. Fig. 2 shows the most important interfaces of the Core DOM using Web IDL [22], a formal notation used in the standard [24]. For each class (expressed as an interface in Web IDL), the Core DOM formalization introduces a HOL datatype representing a typed pointer. For example, the pointer for `Node` is modeled as:

```
datatype 'node_ptr node_ptr = Ext 'node_ptr
```

Here, object-oriented sub-typing (inheritance) is modeled using the type variable `'node_ptr`. Classes are HOL records, e. g.,

```
record Node = Object + nothing :: unit
```

where we see that `Node` inherits from `Object`.

The Core DOM formalization provides an object-oriented data model of a tree-like data structure, which is called *node tree* in the DOM standard, where 1. the root of the tree is an instance of `Document`, 2. instances of the

```
interface Document {
 readonly attribute
          DocumentType? doctype;
 readonly attribute
          Element? documentElement;
};

interface Node {
 readonly attribute
          Document? ownerDocument;
 readonly attribute
          Node? parentNode;
};
```

```
interface CharacterData : Node {
  attribute DOMString data;
};
interface Element : Node {
 readonly attribute
          DOMString tagName;
 readonly attribute
          NodeList childNodes;
 readonly attribute
          NamedNodeMap attributes;
 readonly attribute
          ShadowRoot? shadowRoot;
};
```

Fig. 2. The interface specification of the Core DOM.

class `Element` can be internal nodes or leaves, and 3. instances of the class `CharacterData` can only appear as leaves. Moreover, the Core DOM formalization defines a heap for "storing documents," i. e., instances of the DOM data model. A DOM heap is a finite map from object pointers to objects:

```
datatype ('object_ptr, 'Object) heap
    = Heap (the_heap: ('object_ptr object_ptr, 'Object Object) fmap)
```

On top of this data model, the Core DOM formalization also defines methods for creating, querying, and modifying DOM heaps:

- `get_attribute` returns the attribute (e. g., `id` or `class`) of an element,
- `set_attribute` sets the attribute of an element,
- `get_tag_name` returns the tag type (e. g., `div`) of an element,
- `set_tag_name` sets the tag type of an element,
- `get_child_nodes` returns the children of an element or the document element of a document,
- `get_ancestors` returns a list of ancestor nodes, with the first node being the argument itself, the second one being the parent, and so on,
- `get_parent` returns the parent.
- `get_root_node` returns the root node, the node that is obtained after repeatedly calling `get_parent`,
- `insert_before` inserts the given node into the children of the argument, possibly removing it first from its former parent,
- `get_element_by_id` traverses the tree in depth-first pre-order (called tree-order in the standard) and returns the first element matching the given id,
- `get_owner_document` returns the owner document.

The formal signatures of these methods in Isabelle/HOL are: Fig. 3 provides an overview of the formal signatures in Isabelle/HOL: All methods return a program of type `(_, 'result) dom_prog`, where `'result` can be interpreted as the "real" return type of the methods. A `dom_prog` takes a heap and returns either an error or a result along with a new heap.

```
get_attribute :: (_) element_ptr ⇒ attr_key
                 ⇒ (_, attr_value option) dom_prog
set_attribute :: (_) element_ptr ⇒ attr_key ⇒ attr_value option
                 ⇒ (_, unit) dom_prog
get_tag_name :: (_) element_ptr ⇒ (_, tag_type) dom_prog
set_tag_name :: (_) element_ptr ⇒ tag_type ⇒ (_, unit) dom_prog
get_child_nodes :: (_) object_ptr ⇒ (_, (_) node_ptr list) dom_prog
get_ancestors :: (_::linorder) object_ptr
                  ⇒ (_, (_) object_ptr list) dom_prog
get_parent :: (_) node_ptr ⇒ (_, (_) object_ptr option) dom_prog
get_root_node :: (_) object_ptr ⇒ (_, (_) object_ptr) dom_prog
insert_before :: (_) object_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog
get_element_by_id :: (_) object_ptr ⇒ attr_value
                      ⇒ (_, (_) element_ptr option) dom_prog
get_owner_document :: (_) object_ptr ⇒ (_, (_) document_ptr) dom_prog
```

Fig. 3. The methods for creating, querying, and modifying the DOM. All functions return a program of type `(_, 'result) dom_prog`, where `'result` can be interpreted as the "real" return type of the function. A `dom_prog` takes a heap and returns either an error or a result and a new heap.

Not all objects in a heap are necessarily a regular node of a DOM instance. The DOM also introduces the concept of *disconnected* nodes (e. g., for freshly created objects or local variables), which are unreachable from the main document until they are inserted into the node tree by, e. g., using `insert_before`.

3 Motivating Example: Fancy Tab

In this section, we discuss our running example Fancy Tab from Fig. 1. Fig. 4a focuses on the HTML part of defining Fancy Tab. As the DOM standard does not allow for creating shadow roots statically (i. e., using pure HTML), the definition of shadow roots requires JavaScript to create them at run-time. In our example, we assign the actual definition to `innerHTML` of an already created shadow root.

Fig. 4b shows an attempt to provide the functionality of Fancy Tab *without* using shadow roots. While this alternative definition provides—at first glance—a similar "look and feel," it does not provide any form of run-time separation.

For both variants, we assume that a web developer consumes Fancy Tab without being familiar with its implementation and would like to style their navigation buttons by changing the label text to upper case. Let us assume that the web developer uses the JavaScript snippet from Fig. 5c to change the button texts, which traverses the document's node tree starting from its root node, looking for buttons, and changing their `innerText` attribute.

Now, let us observe the results: Fig. 5b shows the version without shadow roots; here, all buttons, including the navigation buttons on the bottom (which belong to the widget and should be abstracted away), turned upper case, as

```
shadowRoot.innerHTML = '
  <style>...</style>
  <div id="tabs">
    <slot id="tabsSlot" name="title"></slot>






  </div>
  <div id="panels">
    <slot id="panelsSlot"></slot>









  </div>
  <button id="left">
    Previous Tab</button>
  <button id="right">
    Next Tab</button>
';
```

(a) Excerpt of the source of the Fancy Tab widget. We assign the HTML definition to the `innerHTML` of an already created shadow root.

```
<div>
  <style>...</style>
  <div id="tabs">
    <button slot="title">
      Politics</button>
    <button slot="title" selected>
      Sports</button>
    <button slot="title">
      Culture</button>
  </div>
  <div id="panels">
    <section>content panel 1</section>
    <ul><li>News Item 1
          <button>Share</button></li>
        <li>News Item 2
          <button>Share</button></li>
        <li>News Item 3
          <button>Share</button></li></ul>
    <section>
      content panel 3</section>
  </div>
  <button id="left">
    Previous Tab</button>
  <button id="right">
    Next Tab</button>
</div>
```

(b) Defining Fancy Tab without shadow roots would require mixing the code of Fancy Tab and the consuming app, *losing any kind of separation properties.*

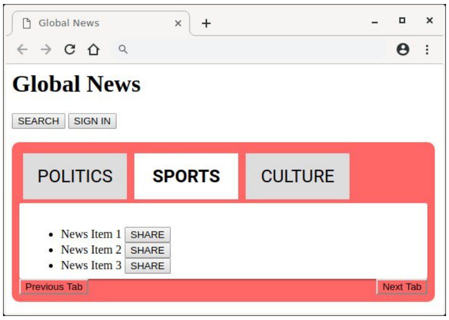
Fig. 4. The source code of Fancy Tab with shadow roots (left) and without (right).

they are part of the same scope. We consider this undesired behavior, because the developer inadvertently modified the internal representation of Fancy Tab: we would like the Fancy Tab developer to be protected from these kinds of effects.

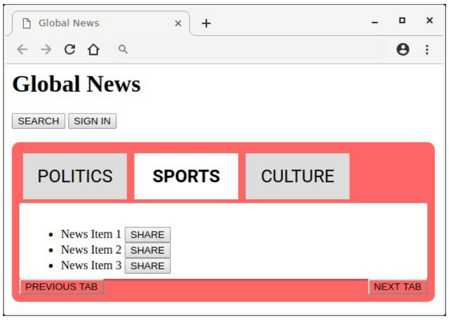
Fig. 5a shows the version with shadow roots, where we can see that only the buttons in the top navigation bar turned upper case—the navigation buttons on the bottom remain unaffected, because they are not part of the same scope (i. e., they are not part of the `document`). To understand the difference, we need to look at the DOM representation with shadow roots as shown in Fig. 6; by calling `document.getElementsByTagName("button")`, we enumerate all buttons, starting from the root document, and thus traverse the tree along the solid arrows. The method `getElementsByTagName` traverses the tree in depth-first pre-order (called *tree order* in the DOM standard), which does not descend along the dotted line.

4 Formalizing Shadow Trees

In this section, we describe how we formalize the addition of shadow trees, i. e., sub-trees of a DOM whose root node is a shadow root, to the DOM. In particular,



(a) Styling Fancy Tab with shadow root only affects buttons outside of Fancy Tab, but not inside.



(b) Styling fancy tabs without shadow root affects additionally buttons inside of Fancy Tab.

```
for (let btn of document.getElementsByTagName("button")) {
   btn.innerText = btn.innerText.toUpperCase();
}
```

(c) A simple JavaScript snippet that converts all button labels to upper case.

Fig. 5. Difference of modifying a website with shadow roots and one without.

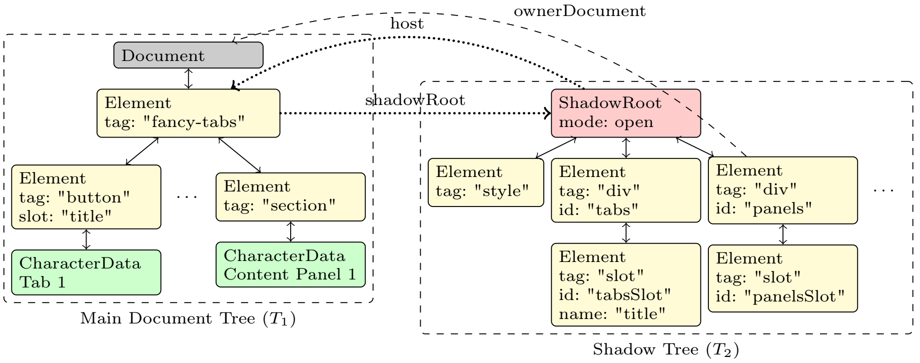


Fig. 6. Representation of the internal DOM structure of our running example Fig. 1.

we 1. extend the data model of the Core DOM to support shadow roots, 2. elicit and formalize invariants that are not made explicit in the DOM standard, and 3. formalize methods for querying and modifying shadow roots.

4.1 Data Model and Basic Accessors

To represent shadow trees, we introduce a new type: `ShadowRoot`. Using Web IDL, the interface of a shadow root is given as:

```
interface ShadowRoot {
  readonly attribute ShadowRootMode mode;
  readonly attribute Element host;
  readonly attribute NodeList childNodes;
}
```

Shadow roots can only be contained in an `Element`, where they behave as a special kind of child (successor) node. Instead of using `get_child_nodes` or `get_parent`, we will introduce the methods `get_shadow_root` and `get_host`, respectively, that act similarly. We formalize these signatures as follows:

```
get_shadow_root ::
    (_) element_ptr ⇒ (_, (_) shadow_root_ptr option) dom_prog
get_mode :: (_) shadow_root_ptr ⇒ (_, shadow_root_mode) dom_prog
get_host :: (_) shadow_root_ptr ⇒ (_, (_) element_ptr) dom_prog
```

Shadow roots manage a list of children, like an `Element` does, but also a flag `mode` that indicates whether the sub-tree shall be accessible from the outside. For this purpose, this flag affects methods such as `get_shadow_root`, i. e., they will not return any nodes if the shadow tree is closed. Fig. 6 illustrates how the new node fits into the concept of a tree structure as defined by the Core DOM.

4.2 Tree Order and DOM Invariants

The data model given in the DOM standard describes a directed object graph, but not necessarily a tree-like data structure. The fact that a valid DOM needs to be a tree-like data structure is only given implicitly: The standard informally defines the concept of a *tree order* as "pre-order depth-first search." The tree order is defined in two variants: *(shadow-excluding) tree order* and *shadow-including tree order*. The former ignores shadow roots while the latter traverses (open) shadow roots prior to traversing the "regular" child nodes. We formalize the shadow-including tree order as follows:

```
partial_function (dom_prog) to_tree_order_si
  :: (_) object_ptr ⇒ (_, (_) object_ptr list) dom_prog where
   to_tree_order_si ptr = do {
    children ← get_child_nodes ptr;
    shadow_root_part ← (case cast ptr of
      Some element_ptr ⇒ do {
        shadow_root_opt ← get_shadow_root element_ptr;
        (case shadow_root_opt of
          Some shadow_root_ptr ⇒ return [cast shadow_root_ptr] |
          None ⇒ return [])
      } |
      None ⇒ return []);
    treeorders ← map_M to_tree_order_si
      ((map cast children) @ shadow_root_part);
    return (ptr # concat treeorders)
  }
```

While not explicitly stated by the standard, it implicitly assumes that the algorithm computing a shadow-including tree order terminates for all "valid" instances of the DOM. In our formalization, this is expressed as the property that "for all well-formed heaps, the (partial) function `to_tree_order_si` does not produce an error." In the following, we will discuss the requirements that are necessary to formally prove this property.

A well-formed heap needs to fulfill several properties that can either be modeled as typing constraints or predicates. An example for the former is the property "an element has at most one attached shadow root," which is, both in the DOM standard and our formalization, enforced by the type system. As an example for the latter, we model the requirement that "shadow roots in a node tree are always attached to a valid host" by using a predicate:

```
definition shadow_root_valid :: (_) heap ⇒ bool where
  shadow_root_valid h = (∀shadow_ptr |∈| fset (shadow_root_ptr_kinds h).
    (∃host. host |∈| element_ptr_kinds h ∧
      |h ⊢ get_tag_name host|r ∈ safe_shadow_root_element_types ∧
      |h ⊢ get_shadow_root host|r = Some shadow_ptr))
```

The constraint `shadow_root_is_valid` is described in the standard informally as requirement that a DOM does not contain "disconnected" shadow roots.

Moreover, the standard required that shadow roots cannot belong to more than one host:

```
definition distinct_lists :: (_) heap ⇒ bool where
  distinct_lists h = distinct (concat (
    map (λelement_ptr. (case
      |h ⊢ get_shadow_root element_ptr|r of
        Some shadow_root_ptr ⇒ [shadow_root_ptr] |
        None ⇒ []))
    |h ⊢ element_ptr_kinds_M|r
  ))
```

To ensure that the DOM with shadow roots is a tree-like data structure, we need to ensure that the underlying object-graph is acyclic. In HOL, we model this in two steps. First, we define a relation between hosts and shadow roots:

```
definition host_shadow_root_rel :: (_) heap ⇒
    ((_) object_ptr × (_) object_ptr) set where
  host_shadow_root_rel h = (λ(x, y).
    (cast x, cast y)) ` {(host, shadow_root).
      host |∈| element_ptr_kinds h
      ∧ |h ⊢ get_shadow_root host|r = Some shadow_root}
```

This relation captures the requirement that the "link" between shadow roots and hosts is a reversible relation. Second, we make use of the pre-defined `acyclic` predicate of HOL for arbitrary relations to postulate that this relation is acyclic.

Now, we can formally capture the concept of a well-defined heap:

```
definition heap_wf :: (_) heap ⇒ bool where
  heap_wf h ⟷
  Core_DOM.heap_wf h ∧
     acyclic (Core_DOM.parent_child_rel h ∪ host_shadow_root_rel h)
     ∧ all_ptrs_in_heap h ∧ distinct_lists h ∧ shadow_root_valid h
```

More precisely, a well-defined heap requires that the regular parent-child relation (defined in the Core DOM [6]) *together* with the shadow root-host relation is acyclic, so we combine them and need `parent_child_rel h ∪ host_shadow_root_rel h` to be acyclic. Also, we require that all pointers in a DOM instance are pointing to instances that are members of the well-defined heap (this is captured by `all_ptrs_in_heap h`).

Additionally, we introduce a short-hand predicate `valid_heap` (which we will use in lemmas throughout this paper) that captures `heap_wf`, but also ensures that the heap only contains pointers and objects whose types correspond (`type_wf`) and pointers that are "known" (`known_ptrs`), which is a property related to the extensibility (see [5, 8] for details on how to encode extensible object-oriented data models in HOL) of the formal model:

```
definition valid_heap :: (_) heap ⇒ bool where
  valid_heap h = heap_wf h ∧ type_wf h ∧ known_ptrs h
```

We can now formally prove, in Isabelle/HOL, that `to_tree_order_si` will always terminate for well-defined heaps, meaning its execution is error-free (captured by the predicate `ok`):

```
lemma to_tree_order_si_ok:
  assumes valid_heap h
  assumes ptr |∈| object_ptr_kinds h
  shows h ⊢ ok (to_tree_order_si ptr)
```

This ensures termination, since `to_tree_order_si` is a partial functions in Isabelle/HOL that maps the case of non-termination to a value of our error type.

5 Web Components

We will now focus on the semantics of web components. While the DOM standard introduces the API for working with shadow trees, it neither defines the concept of a component nor specifies the safety guarantees that should be provided to authors or consumers of components.

5.1 A Formal Definition of Web Components

Many DOM methods, e. g., `get_element_by_id`, traverse the node tree top-down exclusively along the `childNodes` relation (i. e., in shadow-excluding tree order). These methods will not traverse the DOM along the `shadowRoot` relation.

Similarly, methods, such as `get_root_node` only traverse the tree bottom-up using the `parent` relation; they will not continue along the `host` relation.

Intuitively, the `shadowRoot` relation acts as a "component boundary" that can only be crossed by explicitly calling a method that is defined to traverse the `shadowRoot-host` relation.

The standard informally introduces a *(shadow-excluding) tree order* computation (in the standard this is an abstract concept, i. e., not a method available directly to web developers) that returns, in depth-first pre-order, all nodes reachable from a given node by traversing the `childNodes` relation. We use its formalization `to_tree_order` to provide a formal definition of web components:

Definition 1 (Web Component). *A* (Web) Component *of an object o is defined as the list of all objects in* tree order *that are reachable from the root node of o. Formally, we define:*

```
definition
  get_component :: (_) object_ptr ⇒ (_, (_) object_ptr list) dom_prog
where
  get_component ptr = get_root_node ptr ⤜ to_tree_order
```

Informally, an object o belongs to a component c if and only if o is in the list of nodes that are reachable from the root of c via the `childNodes` relation. In our running example (see Fig. 6) the set of all objects is divided into two components: T_1 and T_2.

Our component definition naturally allows distinguishing three different types of components, based on the type of their root node.

Definition 2 (Document Component). *A* Document Component *is a web component where the root node is of type* `Document`*. Formally, we define:*

```
definition is_document_component :: (_) object_ptr list ⇒ bool
  where is_document_component c = is_document_ptr_kind (hd c)
```

Since an object of type `Document` can only occur as the root node of a node tree, a document component can be considered the main part of a node tree.

Definition 3 (Shadow Root Component). *A* Shadow Root Component *is a web component where the root node is of type* `ShadowRoot`*. Formally, we define:*

```
definition is_shadow_root_component :: (_) object_ptr list ⇒ bool
  where is_shadow_root_component c = is_shadow_root_ptr_kind (hd c)
```

A shadow root component might be considered the "canonical component." It encapsulates its contained nodes from outside components and uses slots to interact with the outer component.

Finally, we define a disconnected component as a component only containing *disconnected nodes* (recall Sect. 2), i. e., nodes that are not reachable by traversing the DOM (not even in shadow-including tree order) from its `ownerDocument`.

Definition 4 (Disconnected Component). *A* Disconnected Component *is a web component where the root node is of type* `Node`*. Formally, we define:*

```
definition is_disconnected_component :: (_) object_ptr list ⇒ bool
  where is_disconnected_component c = is_node_ptr_kind (hd c)
```

Disconnected components will not take part in the rendering of the final node tree. Usually, disconnected components will contain freshly crated object graphs that will become a part of a "regular" DOM instance by passing them as argument to methods such as `append_child`.

5.2 Component Safety

Web components should provide a certain form of safety guarantee to both component developers and consumers of components. Informally speaking, neither should a DOM method unintentionally modify the consuming web application when called in the context of the component, nor the other way round. This is particularly important for web components that are developed in JavaScript, a language without (static) typing and with concepts that support the run-time extension of classes using prototype inheritance.

To address this issue, we introduce the notion of component safety for DOM methods that captures which part of a DOM can be modified by a method. We distinguish three types of methods; ones that

1. only operate within the components of their arguments, as one could argue that it is expected that most methods only operate within their proximity. We will call these methods *strongly component-safe.*
2. only operate within the components of their arguments and any newly created components. While these methods operate outside their perceived boundaries, they at least leave other, existing components untouched. We will call these methods *weakly component-safe.*
3. operate on arbitrary parts of a DOM instance. We will call these methods *unsafe.*

In the following, we will introduce our different levels of component safety. In our formalization, we analyze the level of component safety for all methods of the DOM standard. Due to the limitations of using a shallow embedding in HOL, we cannot provide a single HOL predicate that captures the level of component safety. Instead, we will provide two predicates (one for strong safety, one for weak safety) that capture the essence of the safety definitions, along with a proof pattern that we apply to each DOM method.

We start by defining strong component safety, followed by discussing the formal proofs for showing strong component safety.

Definition 5 (Strong Component Safety). *A DOM method is* strongly component safe *if and only if it does not create, delete, return, or modify any objects outside of the components given by its arguments.*

In HOL, we define this property as follows:

```
definition is_strongly_component_safe :: (_) object_ptr set ⇒
   (_) object_ptr set ⇒ (_) heap ⇒ (_) heap ⇒ bool where
 is_strongly_component_safe S_arg S_result h h' =
   let outside_ptrs  = fset (oeject_ptr_kinds h)  -
       (⋃ptr ∈ S_arg. set |h  ⊢ get_component ptr|_r) in
   let outside_ptrs' = fset (object_ptr_kinds h') -
       (⋃ptr ∈ S_arg. set |h' ⊢ get_component ptr|_r) in
   outside_ptrs = outside_ptrs' ∧
   S_result ∩ outside_ptrs = {} ∧
   (∀outside_ptr ∈ outside_ptrs. preserved (get_M outside_ptr id) h h')
```

The predicate takes the set of pointers that are arguments of the method invocation, the set of pointers that is returned by the method invocation, and the state of the heap before and after the method invocation. It then builds the set of pointers that lie outside of the arguments' components before and after the method invocation, and then makes three assertions: both sets of pointers must be equal, i. e., no pointers have been created or deleted; the intersection of the result pointer set and the set of pointers of the arguments' components must be empty; and all pointers outside of these components must remain unmodified (`get_M outside_ptr id` returns the whole object, which is compared to other objects by comparing all their fields).

We will show how this predicate is used to show the strong component safety of a DOM method. For this purpose, we will look at the proofs of component safety for `get_child_nodes` and `get_element_by_id`, which are both *strongly* component safe. The complete formal proofs for these and all other supported DOM methods are included in our Isabelle formalization. In order to show that `get_child_nodes` is strongly component safe, we prove the following lemma:

```
lemma get_child_nodes_strongly_component_safe:
  assumes valid_heap h
  assumes h ⊢ get_child_nodes ptr →_r children
  assumes h ⊢ get_child_nodes ptr →_h h'
  shows is_strongly_component_safe {ptr} (cast ` set children) h h'
```

The first argument of `is_strongly_dom_component_safe` is S_{arg}, which is the set of all pointers that are arguments to the DOM method call in question—in the case of `get_child_nodes`, that is only `ptr`. The second argument is S_{result}, the set of all pointers that are returned by the method, in this case the returned `children` after they have been appropriately cast to object pointers. The last two arguments, `h` and `h'` refer to the heap states before and after the method call. For `get_child_nodes` they will both be the same, so all that is to show is that none of the `children` are outside of the component of `ptr`. Since the component is constructed by iteratively invoking `get_child_nodes` (`to_tree_order` in Def. 1), it follows that the `children` are indeed inside that component and therefore not outside of it.

Methods that iterate in shadow-excluding tree order are also strongly component safe, as in the case of `get_element_by_id`:

```
lemma get_element_by_id_is_strongly_component_safe:
  assumes valid_heap h
  assumes h ⊢ get_element_by_id ptr id →r Some result
  assumes h ⊢ get_element_by_id ptr id →h h'
  shows is_strongly_component_safe {ptr} {cast result} h h'
```

This is the variant for the case that such an element pointer, `result`, is indeed found. The case that no such element is found is a separate lemma and trivial to prove. The proof idea for this kind of lemma is similar to proof idea for `get_child_nodes`; any object has the same root as its parent, and any node found by `get_element_by_id` has the same root as the anchored object—from the definition of `get_component` it then follows that they also have the same component. As these methods do not modify the heap, this is all we need to show for strong component safety.

We continue by defining weak component safety:

Definition 6 (Weak Component Safety). *A DOM method is* weakly component safe *if and only if it does not delete, return, or modify any objects outside of the components given by its arguments.*

The only difference between weak component safety and strong component safety (Def. 5) is that weak safety allows for the creation of new objects outside of the given components. Clearly, strong component safety implies weak component safety, i. e., any strongly component safe method is also weakly component safe.

In general, methods that create new objects are weakly component safe, because most of them return a new object which is not part of any component yet, thus effectively creating a new one.

Examples of weakly component safe methods are `create_element` (creating a new disconnected component), `create_character_data` (creating a new disconnected component), `create_document` (creating a new document component), and `attach_shadow_root` (creating a new shadow root component). For example, for `create_element` we prove the following lemma:

```
lemma create_element_is_weakly_component_safe:
  assumes valid_heap
  assumes h ⊢ create_element document_ptr tag →r result →h h'
  shows is_weakly_component_safe {cast document_ptr} {cast result} h h'
```

The proof idea is that the only object (that existed in `h`) that is modified is referenced by `document_ptr`, which has the newly created element added to its list of disconnected nodes. The new element pointer forms its own, new component, as it does not belong to the one of `document_ptr` or any other one. This is allowed by the definition of weak component safety.

Now we know that `create_element` is weakly component safe, but we would also like to show that it is indeed *not* strongly component safe. In order to do so, we will again leverage our `is_strongly_component_safe` predicate, but invert our proof pattern:

```
lemma create_element_not_strongly_component_safe:
  obtains h and h' and document_ptr and new_element_ptr and tag
where
  valid_heap h and
  h ⊢ create_element document_ptr tag →r new_element_ptr →h h' and
  ¬ is_strongly_component_safe
      {cast document_ptr} {cast new_element_ptr} h h'
```

The structure of this kind of lemma is different from the lemmas showing safety; we use the Isar **obtains** concept to show that there exists at least one heap for which the method is not strongly component safe. It therefore suffices to construct a counter-example, which in the case of `create_element` can be as small as a heap containing nothing but a single `document_ptr`. Since our model is completely executable, we can use the symbolic execution engine of Isabelle to show that this heap indeed fulfills the lemma.

The next class of methods includes ones that concern shadow roots, which are expectedly generally unsafe. In case of a closed shadow tree (`mode` is set to `Closed`), methods trying to look inside Shadow Root Components (e. g., `get_shadow_root` and `assigned_slot`) will return an error, making these methods strongly component-safe in this case—methods trying to break out (e. g., `get_host`, `get_composed_root_node`, and `get_assigned_nodes`) are not affected by the mode and thus remain unsafe. In the case of `assigned_slot`, this looks as follows:

```
lemma assigned_slot_not_weakly_dom_component_safe:
  obtains h and node_ptr and slot_opt and h'
where
  valid_heap h and
  h ⊢ assigned_slot node_ptr →r slot_opt →h h' and
  ¬ is_weakly_component_safe
     {cast node_ptr} (cast ` set_option slot_opt) h h'
```

We use the same construction as we did for showing that `create_element` was not strongly safe, but now use `is_weakly_component_safe`. We use the function `set_option` to convert the return value of `assigned_slot` into a pointer set, so we can allow both possible outcomes of the method call, regardless of whether a slot has been found. Recall that we are constructing a counter example here, so we only need to find one valid instantiation of variables. The proof follows the usual schema, though the counter-example is more complex than before. We need to create a heap that contains an element, a shadow root, slots and slotables, but otherwise does not require any special configuration.

5.3 Component Safety of the DOM Methods

In the following, we will discuss to what extent the methods defined in the DOM standard are component safe. By doing this we effectively evaluate how suitable shadow roots are for providing separation. We will see that some methods, in particular `append_child`, break our separation in unexpected ways.

Table 1 summarizes our classification. All shown lemmas with their proofs can be found in our formalization (in the formalization, the lemma names start with the method name, followed by `_is` or `_not`, followed either by the suffix `_strongly_component_safe`, `_weakly_component_safe`, or, if the method is not safe at all, `_component_unsafe`). The two methods `get_shadow_root` and `assigned_slot` are only safe if the DOM only contains closed shadow roots.

Table 1. Classification of the DOM methods into whether they are strongly or weakly component safe, or not at all. The last column (closed) classifies the methods for the special case that the DOM instance only contains closed shadow roots.

Method	Component Safety	
	open	closed
`get_child_nodes`	strong	strong
`get_parent`	strong	strong
`get_root_node`	strong	strong
`get_element_by_id`	strong	strong
`get_elements_by_class_name`	strong	strong
`get_elements_by_tag_name`	strong	strong
`create_element`	weak	weak
`create_character_data`	weak	weak
`create_document`	weak	weak
`attach_shadow_root`	weak	weak
`get_shadow_root`	unsafe	safe
`get_host`	unsafe	unsafe
`get_composed_root_node`	unsafe	unsafe
`get_assigned_nodes`	unsafe	unsafe
`assigned_slot`	unsafe	safe
`adopt_node`	unsafe	unsafe
`remove_child`	unsafe	unsafe
`insert_before`	unsafe	unsafe
`append_child`	unsafe	unsafe
`get_owner_document`	unsafe	unsafe

From the view of a web developer, the fact that `get_owner_document` is unsafe is particularly worrisome: if the root node of a given pointer is not a document, then this method will return a document that is outside of the current component. Thus, if a library developer uses this method for setting up their component, they might inadvertently break out and change arbitrary objects outside of their component.

Surprisingly and unfortunately, many of the heap-modifying methods such as `adopt_node`, `remove_child`, `insert_before`, and `append_child` are unsafe, too, because they all access and modify the list of disconnected nodes of owner documents. For example, if an element is removed by using `remove_child`, it gets added to the list of disconnected nodes of the argument's owner document, which is outside of the component of the removed child.

We have seen that there are a number of DOM methods which we could prove to be unsafe with regard to components. This is undesirable, as this means that these methods break the expectations that a developer might have when working with shadow root components.

5.4 Recommendations

Web components based on shadow trees are an important step forward for a component-based web development approach. They allow web developers to define components with well-defined interfaces (called slots) for interacting with the embedding application or other components (components can be nested arbitrarily). However, our formal analysis shows that there are subtle ways to accidentally break the component boundaries: most prominently, the enclosing owner document is easily accessible from inside a shadow root component by using the `ownerDocument()` method on any node of that component, which corresponds to the ubiquitous `document` reference in any (Web) JavaScript context. We suggest changing this behavior and instead providing a reference to the root of the current component, thus strengthening the component separation against accidental interference with other components. This would, on the one hand, remove the most unexpected way of breaking up the component boundaries and, on the other hand, simplify the overall definition of web components. This change would also simplify the notion of component safety by removing boundary cases for disconnected nodes.

A second point of concern is that methods such as `remove_child` can have unexpected effects outside of shadow trees, even if all arguments lie within that shadow tree. Such removed nodes get added to the context of the surrounding document, from which they might added again onto other, unrelated components of the same document (DOM instance).

6 Related Work

To the best of our knowledge, we are the first to formalize the concept of shadow roots. The most closely related works are our own formalization of the DOM [7] *without* shadow roots that we use as basis of our component model, and the works of Gardner et al. [12], Raad et al. [19], Smith [20]. In the latter ones, the authors present a non-executable, non-extensible, and non-mechanized operational semantics of a minimal DOM and show how this semantics can be used for Hoare-style reasoning for analysis heaps of DOMs. The authors focus on providing a formal foundation for reasoning over client-side JavaScript programs that modify the DOM. Neither of these works defines formally the concept of web components nor do they define component safety or formally analyze the behavior of DOM methods in the context of shadow trees.

Our work shares a common goal with ownership type systems [10]. For example, Poetzsch-Heffter et al. [18] use type annotations to give objects in object-oriented programs a notion of ownership. This enables them to allow certain components only read-access to an object, while the owner might have full read and write-access. This line of work is orthogonal to ours; it is certainly possible to create an access-control layer on top of our web components, but we are more concerned with components inside a tree-like structure and how a given set of methods behave regarding the boundary induced by shadow roots.

A more informal model of the DOM that focuses on the needs of building a static analysis tool for client-side JavaScript programs is presented by Jensen et al. [15]. This model does not focus on the DOM as such, instead the authors focus on the representation of HTML documents on top of the DOM.

There are also very few formalizations of data structures for manipulating XML-like document structures available. The most closely related one is presented by Sternagel and Thiemann [21]. The authors present an "XML library" for Isabelle/HOL. The purpose of this library is to provide XML parsing and pretty printing facilities for Isabelle. As such, it is not a formalization of XML or XML-like data structures in Isabelle/HOL.

Shadow roots seem to achieve a very similar goal as the `<iframe>`-tag of the HTML standard. Still, the motivation for both differ significantly: while iframes were introduced to allow the *secure* integration of content from different websites, shadow roots were introduced to allow component-based web development similar to, for example, using components in the .net framework. The limitations of shadow roots to ensure the privacy of data processed by web applications have already been discussed by Légaré et al. [16] and Freyberger et al. [11].

Finally, there are several works, e. g., [1, 4, 13, 14] on formalizing parts of web browsers for analyzing their security. These works use high-level specifications of web browsers and do not contain a formalization of the DOM itself.

7 Conclusion

We present a formal model of web components and component safety, and we formally verify the level of component safety for the DOM API as defined in the DOM standard [24].

Our formalization of the DOM with shadow roots and its API is an important step towards providing formal guarantees for a modular development approach to web applications as well as increasing the security and safety of large web applications. While the current proposal clearly has weaknesses, our analysis also shows that moderate changes to the concept of shadow roots can make the web components a much more powerful and stronger concept that, hopefully, also can make developing secure applications easier.

On a technical level, our formalization is based on a shallow embedding of the DOM with shadow roots into Isabelle/HOL. We use only conservative extensions of HOL (i. e., we do not introduce any axioms). Hence, our formalization is consistent by construction. Overall, it consists of more than 10 000 lines of Isabelle code, including conservative definitions and proofs. To ensure the compliance of our formalization to the official DOM standard, we aimed for an executable formalization: an executable specification allows for symbolically evaluating the official compliance test suite in Isabelle/HOL. Thus, as our formalization passes the test suite, our formalization adheres to the same compliance standards as the widely used web browser engines.

Future Work. While web components based on shadow roots provide some form of isolation for JavaScript developers, they have weaknesses and, clearly, cannot provide the isolation necessary for enforcing security guarantees similar to iframes. While iframes are a rather old concept that is defined on top of the DOM in the HTML standard [25], shadow roots are a very recent concept that is integrated into the DOM [24]. On the first glance, the two concepts do not have much in common. Having a closer look reveals that the concepts are closely related: on the one hand, it seems desirable to introduce security concepts to shadow roots and, on the other hand, iframes would clearly benefit from interfaces allowing web developers to adapt certain aspects of an included iframe. Thus, the question emerges whether shadow roots can, in the long term, replace iframes. To answer this question, we plan to formalize the core of the HTML standard on top of our DOM formalization. This allows us to compare both concepts formally and also formally investigate the impact of adding security features to shadow roots.

References

[1] Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: IEEE Computer Security Foundations Symposium (CSF), pp. 290–304. IEEE Computer Society (2010). doi: 10.1109/CSF.2010.27.

[2] Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof. 2nd edn. Kluwer Academic Publishers, Dordrecht (2002)

[3] Bidelman, E.: Shadow dom v1: Self-contained web components (2017). `https://developers.google.com/web/fundamentals/getting-started/primers/shadowdom`

[4] Bohannon, A., Pierce, B.C.: Featherweight Firefox: Formalizing the core of a web browser. In: Usenix Web Application Development (WebApps) (2010).

[5] Brucker, A.D.: An interactive proof environment for object-oriented specifications. Ph.D. thesis, ETH Zurich (2007). ETH Dissertation No. 17097.

[6] Brucker, A.D., Herzberg, M.: The Core DOM. Archive of Formal Proofs (2018). `http://www.isa-afp.org/entries/Core_DOM.html`, Formal proof development

[7] Brucker, A.D., Herzberg, M.: A formal semantics of the Core DOM in Isabelle/HOL. In: Champin, P., Gandon, F.L., Lalmas, M., Ipeirotis, P.G. (eds.) The 2018 Web Conference Companion (WWW), pp. 741–749. ACM Press (2018). doi: 10.1145/3184558.3185980.

[8] Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in HOL. Journal of Automated Reasoning **41**, 219–249 (2008). doi: 10.1007/s10817-008-9108-3.

[9] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940)

[10] Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: A survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification, LNCS 7850, pp. 15–58. Springer-Verlag (2013). doi: 10.1007/978-3-642-36946-9_3

[11] Freyberger, M., He, W., Akhawe, D., Mazurek, M.L., Mittal, P.: Cracking shadowcrypt: Exploring the limitations of secure I/O systems in internet browsers. PoPETs **2018**(2), 47–63 (2018). doi: 10.1515/popets-2018-0012

[12] Gardner, P., Smith, G., Wheelhouse, M.J., Zarfaty, U.: DOM: towards a formal specification. In: Programming Language Technologies for XML (PLAN-X), ACM (2008).

[13] Guha, A., Fredrikson, M., Livshits, B., Swam, N.: Verified security for browser extensions. In: IEEE Symposium on Security and Privacy, pp. 115–130 (2011). doi: 10.1109/SP.2011.36

[14] Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: Kohno, T. (ed.) USENIX, pp. 113–128. USENIX (2012).

[15] Jensen, S.H., Madsen, M., Møller, A.: Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In: ESEC/FSE, pp. 59–69. ACM (2011). doi: 10.1145/2025113.2025125.

[16] Légaré, J., Sumi, R., Aiello, W.: Beeswax: a platform for private web apps. PoPETs **2016**(3), 24–40 (2016)

[17] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, LNCS 2283. Springer-Verlag (2002). doi: 10.1007/3-540-45949-9

[18] Poetzsch-Heffter, A., Geilmann, K., Schäfer, J.: Infering ownership types for encapsulated object-oriented program components. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Program Analysis and Compilation, Theory and Practice, pp. 120–144. Springer (2007). doi: 10.1007/978-3-540-71322-7_6

[19] Raad, A., Santos, J.F., Gardner, P.: DOM: specification and client reasoning. In: Igarashi, A. (ed.) Programming Languages and Systems (APLAS), LNCS 10017, pp. 401–422. Springer (2016). doi: 10.1007/978-3-319-47958-3_21

[20] Smith, G.D.: Local reasoning about web programs. Ph.D. thesis, Imperial College London, London, UK (2011)

[21] Sternagel, C., Thiemann, R.: XML. Archive of Formal Proofs (2014). `http://isa-afp.org/entries/XML.shtml`, Formal proof development

[22] W3C: Web IDL (2017). `https://heycam.github.io/webidl/`

[23] W3C: Shadow DOM (2018). `https://www.w3.org/TR/2018/NOTE-shadow-dom-20180301/`. Last Updated 1 March 2018

[24] WHATWG: DOM – living standard (2019). `https://dom.spec.whatwg.org/commit-snapshots/7fa83673430f767d329406d0aed901f296332216/`. Last Updated 11 February 2019

[25] WHATWG: HTML – living standard (2019). `https://html.spec.whatwg.org/commit-snapshots/b8c084e9d5461b858180e7f80ad6ca19c7963723/`. Last Updated 19 February 2019