Formalizing (Web) Standards

An Application of Test and Proof

Achim D. Brucker and Michael Herzberg
{a.brucker, msherzberg1}@sheffield.ac.uk

Software Assurance & Security Research
Department of Computer Science, The University of Sheffield, Sheffield, UK
https://logicalhacking.com/

International Conference on Tests & Proofs 2018
June 27, 2018 Toulouse, France

{*LogicalλHacking*}.com

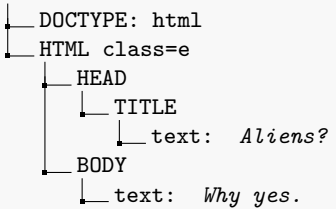
The University Of Sheffield.

Outline

What is the Document Object Model (DOM)?

```
<!DOCTYPE html>
<html class=e>
 <head>
  <title>Aliens?</title>
 </head>
 <body>Why yes.</body>
</html>
```

HTML



DOM

Why yes.

Rendering

Why is the DOM important?

Short answer:

- The DOM is the core data structure used by web browsers

Long(er) answer:

- If the DOM implementation is
  - insecure
  - incorrect

  the whole browser is insecure/incorrect
- Many web security mechanism (e.g., CSP) are defined in terms of access to the DOM:
  - we can formalize aspects of Web security without formalizing JavaScript
  - we can compare novel security/component concepts emerging in browsers
- Many implementations available (for managing tree-structured documents), e.g.,
  - libxml2 (C, bindings for various languages)
  - Xerces (Java, C++, Perl)
  - Saxon XLST (Java, JavaScript, .NET)
  - PHP.Gt DOM (PHP)
  - Domino (Node.js)
  - Edge (e.g., Microsoft Edge Browser)
  - Gecko (e.g., Mozilla Firefox)
  - KHTML (e.g., KDE Konqueror)
  - WebKit, fork of KHTML (e.g., Safari)
  - Chrome, fork of KHTML

The Official Standard

DOM

Living Standard — Last Updated 16 April 2018

Participate:

GitHub whatwg/dom (new issue, open issues)
IRC: #whatwg on Freenode

Commits:

GitHub whatwg/dom/commits
Snapshot as of this commit
@domstandard

Tests:

web-platform-tests dom/ (ongoing work)

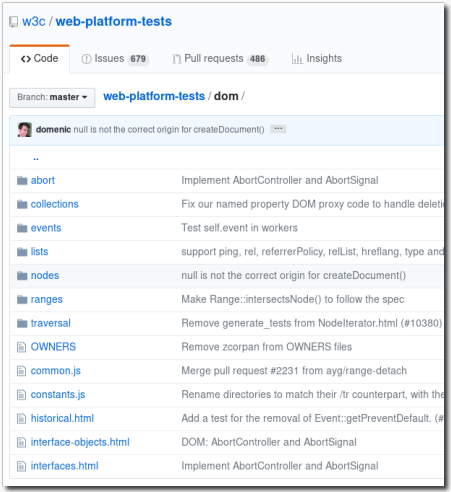
Translations (non-normative):

日本語

Abstract

DOM defines a platform-neutral model for events, aborting activities, and node trees.

Table of Contents



w3c / web-platform-tests

Code | Issues 679 | Pull requests 486 | Insights

Branch: master ▾ web-platform-tests / dom /

domenic null is not the correct origin for createDocument() ···

..	
abort	Implement AbortController and AbortSignal
collections	Fix our named property DOM proxy code to handle deleti
events	Test self.event in workers
lists	support ping, rel, referrerPolicy, relList, hreflang, type and
nodes	null is not the correct origin for createDocument()
ranges	Make Range::intersectsNode() to follow the spec
traversal	Remove generate_tests from NodeIterator.html (#10380)
OWNERS	Remove zcorpan from OWNERS files
common.js	Merge pull request #2231 from ayg/range-detach
constants.js	Rename directories to match their /tr counterpart, with the
historical.html	Add a test for the removal of Event::getPreventDefault. (#
interface-objects.html	DOM: AbortController and AbortSignal
interfaces.html	Implement AbortController and AbortSignal

The Official Standard

```
[CEReactions] Node insertBefore(Node node, Node? child);
```

The **insertBefore(*node*, *child*)** method, when invoked, must return the result of pre-inserting *node* into context object before *child*.

To **pre-insert** a *node* into a *parent* before a *child*, run these steps:

1. Ensure pre-insertion validity of *node* into *parent* before *child*.
2. Let *reference child* be *child*.
3. If *reference child* is *node*, set it to *node*'s next sibling.
4. Adopt *node* into *parent*'s node document.
5. Insert *node* into *parent* before *reference child*.
6. Return *node*.

```
test(function() {
  var a = document.createElement('div');
  var b = document.createElement('div');
  var c = document.createElement('div');
  assert_throws('NotFoundError', () => {
    a.insertBefore(b, c);
  });


},'Calling␣insertBefore␣with␣a␣reference' +
  'child␣whose␣parent␣is␣not␣the␣context' +
  'node␣must␣throw␣a␣NotFoundError.')
```

Formalizing insertBefore

```
[CEReactions] Node insertBefore(Node node, Node? child);
```

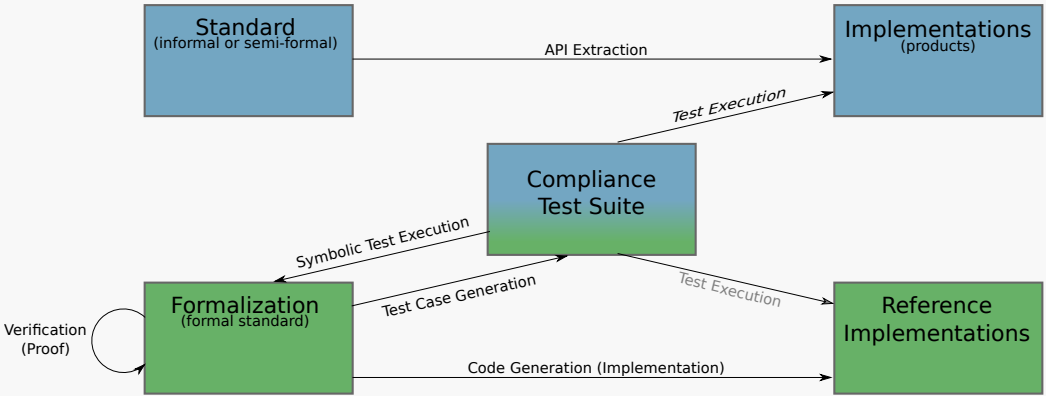
The **insertBefore(*node*, *child*)** method, when invoked, must return the result of pre-inserting *node* into context object before *child*.

To **pre-insert** a *node* into a *parent* before a *child*, run these steps:

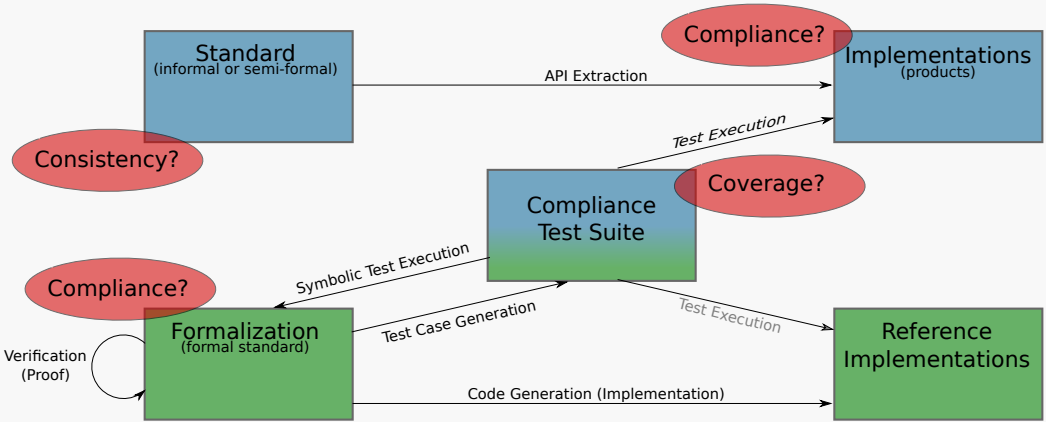
1. Ensure pre-insertion validity of *node* into *parent* before *child*.
2. Let *reference child* be *child*.
3. If *reference child* is *node*, set it to *node*'s next sibling.
4. Adopt *node* into *parent*'s node document.
5. Insert *node* into *parent* before *reference child*.
6. Return *node*.

```
definition insert_before :: "(_) object_ptr
                             ⇒ (_) node_ptr
                             ⇒  node_ptr option
                             ⇒ (_, unit) dom_prog"
  where
    "insert_before ptr node child = do {
      ensure_pre_insertion_validity node ptr child;
      reference_child ←(if Some node = child
        then next_sibling node
        else return child);
      owner_document ←get_owner_document ptr;
      adopt_node owner_document node;
      insert_node ptr node reference_child
    }"
```

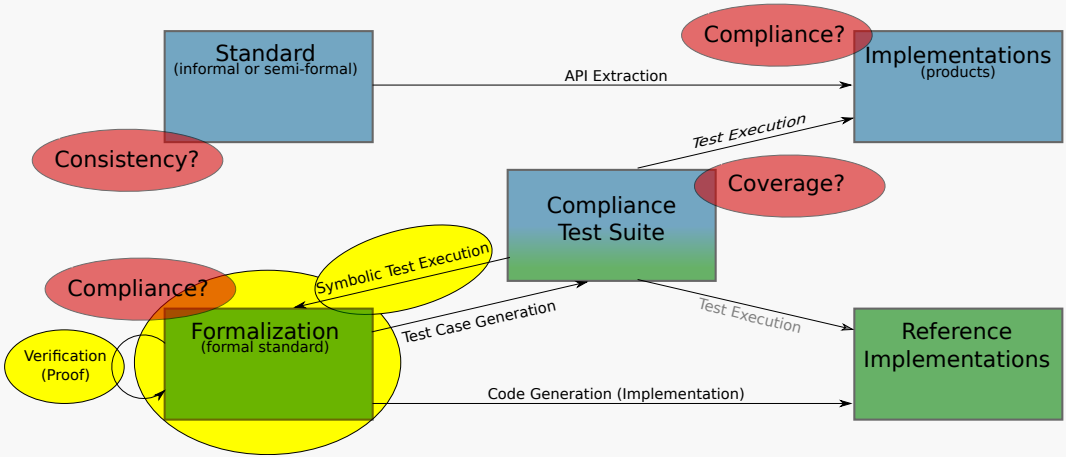
The Benefits of a Formal Standard



The Benefits of a Formal Standard



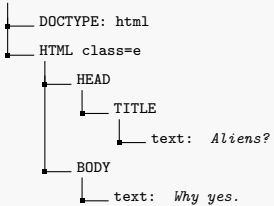
The Benefits of a Formal Standard

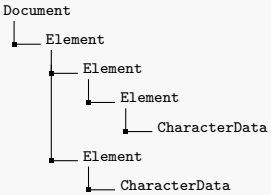


A Simple Functional Tree Datatype?

The node tree seems like it can be modeled by a simple functional tree datatype, with …

- …`Document` as roots
- …`Element` as intermediate nodes
- …`CharacterData` as leaves





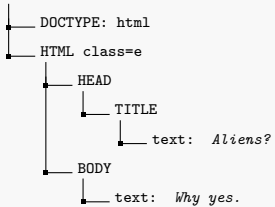
A Simple Functional Tree Datatype?

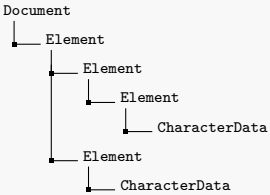
The node tree seems like it can be modeled by a simple functional tree datatype, with …

- …`Document` as roots
- …`Element` as intermediate nodes
- …`CharacterData` as leaves

No, because …

- …functions such as `getParent`
- API is pointer-heavy:






```
[CEReactions] Node insertBefore(Node node, Node? child);
```

Node Tree Properties

Starting with a map as heap, we need to ensure that the heap is actually a tree, meaning …

1. …nodes have maximal one parent
2. …our graph is acyclic
3. …all pointers are actually in the heap (no `NullPointerExceptions`)
4. …the pointer lists are distinct

Node Tree Properties

Starting with a map as heap, we need to ensure that the heap is actually a tree, meaning …

1. … nodes have maximal one parent
2. … our graph is acyclic
3. … all pointers are actually in the heap (no `NullPointerExceptions`)
4. … the pointer lists are distinct

In the standard, all these properties are **implicit**!

A Formal Model in Isabelle/HOL

Highlights of our formal model:

- State-Exception-Monad to allow imperative function definitions
- Way of modeling object-orientation in higher-order logic
- Heap-representation with pointers and objects
- Formal model is executable and OO-extendable

A Formal Model in Isabelle/HOL

Logical definition:

```
record (_) Element = Node +
  tag_type :: tag_type
  child_nodes :: "(_) node_ptr list"
  attrs :: attrs
  shadow_root_opt :: "'shadow_root_ptr shadow_root_ptr option"

definition "get_attribute ptr k = do {m ←get_M ptr attrs; return (m k)}"
```

Recall `insertBefore`

```
[CEReactions] Node insertBefore(Node node, Node? child);
```

The **`insertBefore(node, child)`** method, when invoked, must return the result of pre-inserting *node* into context object before *child*.

To **pre-insert** a *node* into a *parent* before a *child*, run these steps:

1. Ensure pre-insertion validity of *node* into *parent* before *child*.
2. Let *reference child* be *child*.
3. If *reference child* is *node*, set it to *node*'s next sibling.
4. Adopt *node* into *parent*'s node document.
5. Insert *node* into *parent* before *reference child*.
6. Return *node*.

```
definition insert_before :: "(_) object_ptr
                             ⇒ (_) node_ptr
                             ⇒  node_ptr option
                             ⇒ (_, unit) dom_prog"
  where
    "insert_before ptr node child = do {
      ensure_pre_insertion_validity node ptr child;
      reference_child ←(if Some node = child
        then next_sibling node
        else return child);
      owner_document ←get_owner_document ptr;
      adopt_node owner_document node;
      insert_node ptr node reference_child
    }"
```

Question:

Does `insertBefore` preserves distinctness of child nodes?

Example Proof

insertBefore preserves distinctness of child nodes

```
lemma insert_before_children_remain_distinct:
  assumes wellformed: "heap_is_wellformed h"
    and parent_known: "⋀parent. h ⊢get_parent new_child →r Some parent ⟹is_known_ptr parent"
    and known: "is_known_ptr ptr"
    and insert_before: "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children ⟹distinct children"
proof -
  obtain ...
      h': "h ⊢ adopt_node owner_document new_child →h h'" and
      h2: "h' ⊢ insert_node ptr new_child reference_child →h h2"
    by ...
  have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children ⟹distinct children"
    by ...
  moreover have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children"
      "⟹ new_child ∉set children"
    by ...
  ultimately show "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children"
      "⟹ distinct children"
    by ...
qed
```

Example Proof

insertBefore preserves distinctness of child nodes

```
lemma insert_before_children_remain_distinct:
  assumes wellformed: "heap_is_wellformed h"
    and parent_known: "⋀parent. h ⊢get_parent new_child →r Some parent ⟹is_known_ptr parent"
    and known: "is_known_ptr ptr"
    and insert_before: "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children ⟹distinct children"
proof -
  obtain ...
      h': "h ⊢ adopt_node owner_document new_child →h h'" and
      h2: "h' ⊢ insert_node ptr new_child reference_child →h h2"
    by ...
  have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children ⟹distinct children"
    by ...
  moreover have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children"
      "⟹ new_child ∉set children"
    by ...
  ultimately show "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children"
      "⟹ distinct children"
    by ...
qed
```

Example Proof

insertBefore preserves distinctness of child nodes

```
lemma insert_before_children_remain_distinct:
  assumes wellformed: "heap_is_wellformed h"
    and parent_known: "⋀parent. h ⊢get_parent new_child →r Some parent ⟹is_known_ptr parent"
    and known: "is_known_ptr ptr"
    and insert_before: "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children ⟹distinct children"
proof -
  obtain ...
      h': "h ⊢ adopt_node owner_document new_child →h h'" and
      h2: "h' ⊢ insert_node ptr new_child reference_child →h h2"
    by ...
  have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children ⟹distinct children"
    by ...
  moreover have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children"
      "⟹ new_child ∉set children"
    by ...
  ultimately show "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children"
      "⟹ distinct children"
    by ...
qed
```

Example Proof

insertBefore preserves distinctness of child nodes

```
lemma insert_before_children_remain_distinct:
  assumes wellformed: "heap_is_wellformed h"
    and parent_known: "⋀parent. h ⊢get_parent new_child →r Some parent ⟹is_known_ptr parent"
    and known: "is_known_ptr ptr"
    and insert_before: "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children ⟹distinct children"
proof -
  obtain ...
      h': "h ⊢ adopt_node owner_document new_child →h h'" and
      h2: "h' ⊢ insert_node ptr new_child reference_child →h h2"
    by ...
  have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children ⟹distinct children"
    by ...
  moreover have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children"
      "⟹ new_child ∉set children"
    by ...
  ultimately show "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children"
      "⟹ distinct children"
    by ...
qed
```

Example Proof

insertBefore preserves distinctness of child nodes

```
lemma insert_before_children_remain_distinct:
  assumes wellformed: "heap_is_wellformed h"
    and parent_known: "⋀parent. h ⊢get_parent new_child →r Some parent ⟹is_known_ptr parent"
    and known: "is_known_ptr ptr"
    and insert_before: "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children ⟹distinct children"
proof -
  obtain ...
      h': "h ⊢ adopt_node owner_document new_child →h h'" and
      h2: "h' ⊢ insert_node ptr new_child reference_child →h h2"
    by ...
  have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children ⟹distinct children"
    by ...
  moreover have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children"
      "⟹ new_child ∉set children"
    by ...
  ultimately show "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children"
      "⟹ distinct children"
    by ...
qed
```

Example Proof

insertBefore preserves distinctness of child nodes

```
lemma insert_before_children_remain_distinct:
  assumes wellformed: "heap_is_wellformed h"
    and parent_known: "⋀parent. h ⊢get_parent new_child →r Some parent ⟹is_known_ptr parent"
    and known: "is_known_ptr ptr"
    and insert_before: "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children ⟹distinct children"
proof -
  obtain ...
      h': "h ⊢ adopt_node owner_document new_child →h h'" and
      h2: "h' ⊢ insert_node ptr new_child reference_child →h h2"
    by ...
  have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children ⟹distinct children"
    by ...
  moreover have "⋀ptr children. is_known_ptr ptr ⟹h' ⊢get_child_nodes ptr →r children"
      "⟹ new_child ∉set children"
    by ...
  ultimately show "⋀ptr children. is_known_ptr ptr ⟹h2 ⊢get_child_nodes ptr →r children"
      "⟹ distinct children"
    by ...
qed
```

Tests: Our Formal Model Complies with the Standard

Compliance test (JavaScript) from the official suite on Github.

```
test(function() {
  var a = document.createElement('div');
  var b = document.createElement('div');
  var c = document.createElement('div');
  assert_throws('NotFoundError', () => {
    a.insertBefore(b, c);
  });


},'Calling␣insertBefore␣with␣a␣reference' +
  'child␣whose␣parent␣is␣not␣the␣context' +
  'node␣must␣throw␣a␣NotFoundError.')
```

The same test formalized in HOL, using a state-exception-monad.

```
lemma "test (do {
  a ←document.createElement(''div'');
  b ←document.createElement(''div'');
  c ←document.createElement(''div'');
  assert_throws(NotFoundError,
    a.insertBefore(b, c))
}) Node_insertBefore_heap"
  by code_simp

(* 'Calling insertBefore with a reference
  child whose parent is not the context
  node must throw a NotFoundError.' *)
```

And Proofs: Generalizing Test Cases

```
lemma insert_before_reference_child_not_in_children:
  assumes "h ⊢ get_parent child →r Some parent"
    and "ptr ≠ parent"
    and "¬ is_character_data_ptr_kind ptr"
    and "h ⊢ get_ancestors ptr →r ancestors"
    and "cast node ∉ set ancestors"
  shows "h ⊢ insert_before ptr node (Some child) →e NotFoundError"
proof -
  have "h ⊢ ensure_pre_insertion_validity node ptr (Some child) →e NotFoundError"
    using assms unfolding insert_before_def ensure_pre_insertion_validity_def
    by auto (simp | rule bind_returns_error_I2)+
  then show ?thesis
    unfolding insert_before_def by auto
qed
```

Showing Properties in Isabelle Using Test and Proof

Interactive Proofs

- able to show generic properties
- interactive proof (e.g.using induction)
- only small software stack needs to be trusted

Symbolic Execution (code_simp)

- able to show grounded properties
- fully automatic, can be slow for large examples
- only small software stack needs to be trusted

Code Execution (eval)

- able to show grounded properties
- fully automatic, fast
- large software stack needs to be trusted

Conclusion and Future Work

Tests:

- Generation of test cases using HOL-TestGen to
  - improve the compliance test suite
  - compare different implementations wrt their compliance to the standard

Proofs:

- Formalizing an emerging component model (Shadow DOM)
- Formalizing DOM security policies (e.g., Same Origin, CSP)
- Comparing the Shadow DOM to existing security policies (e.g., Same Origin, CSP)

Thank you for your attention!
Any questions or remarks?

Contact:

Dr. Achim D. Brucker and Michael Herzberg
Department of Computer Science
University of Sheffield
Regent Court
211 Portobello St.
Sheffield S1 4DP, UK

{a.brucker, msherzberg1}@sheffield.ac.uk
https://www.mherzberg.de

Bibliography

Achim D. Brucker and Michael Herzberg.

A formal semantics of the core DOM in Isabelle/HOL.

In Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *The 2018 Web Conference Companion* (WWW), pages 741–749. ACM Press, 2018.

Achim D. Brucker and Michael Herzberg.

Formalizing (web) standards: An application of test and proof.

In Cathrine Dubois and Burkhart Wolff, editors, TAP *2018: Tests And Proofs*, number 10889 in Lecture Notes in Computer Science, pages 1–8. Springer-Verlag, 2018.

Document Classification and License Information