Formalizing (Web) Standards
An Application of Test and Proof

Achim D. Brucker and Michael Herzberg

Department of Computer Science, The University of Sheffield, Sheffield, UK
{a.brucker,msherzberg1}@sheffield.ac.uk

Abstract. Most popular technologies are based on informal or semi-formal standards that lack a rigid formal semantics. Typical examples include web technologies such as the DOM or HTML, which are defined by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C). While there might be API specifications and test cases meant to assert the compliance of implementations, the actual standard is rarely accompanied by a formal model that would lend itself for, e. g., verifying the security or safety properties of real systems.
Even when such a formalization of a standard exists, two important questions arise: first, to what extent does the formal model comply with the standard and, second, to what extent does a concrete implementation comply with the formal model and the assumptions made during the verification of certain properties?
In this paper, we present an approach that brings all three involved artifacts—the (semi-)formal standard, the formalization of the standard, and the implementations—closer together by combining verification, symbolic execution, and specification-based testing.

Keywords: Standard compliance · Compliance tests · DOM

1 Introduction

Most popular technologies are only specified by standards using a semi-formal or, worse, an informal notation. Moreover, the tools used for writing standards only support, if at all, trivial consistency checks. Thus, it is no surprise that such standards usually contain inconsistencies (e. g., different sections of the same standard that contradict each other) or unwanted under-specifications (e. g., where the authors of the standard omit the specification of important properties that, e. g., the defined API should fulfill).

Even if a standard is developed formally, or contains a (often non-normative) formalization, two important questions arise: 1. to what extent does the formal model comply with the semi-formal parts of the standard, and 2. to what extent does an actual implementation comply with the formal model? If the formal model was used for verifying properties, one also needs to validate that the real system fulfills the assumptions made during the verification.

C. Dubois and B. Wolff (Eds.): TAP 2018, LNCS 10889, pp. 1–8, 2018.

Neither the problem of glitches and inconsistencies of standards nor the use of testing for showing compliance of implementations are new (see, e. g., [1, 7, 8] for examples of formalizations of standards, outside the Web domain, and the use of testing for showing the compliance of implementations.) Still, most standard development today is based on semi-formal specifications. Prominent examples of such a semi-formal standard development are the standards for the common web technologies, such as the Document Object Model (DOM) or HTML. Both are defined by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C). These web standards are developed in an open process, e. g., everybody can read and comment on upcoming versions of the standard, and they usually include type-checked interfaces for the defined APIs. These interfaces are specified in Web IDL [11]. Moreover, these standards are complemented by a *manually* defined *compliance* test suite that can be used by developers to check their implementation. Additionally, due to the manual process of developing the compliance tests, their quality mostly depends on expert knowledge and their quality varies greatly, depending on who wrote the test cases.

In this paper, we present an approach that brings all three involved artifacts—the standard, the formalization of the standard, and the implementations—closer together by combining verification, symbolic execution, and specification-based testing. Moreover, we report on a case study applying this approach to the Document Object Model (DOM) standard [10, 13] that specifies *the* central data structure of all modern web browsers as well as algorithms for querying and updating the DOM. Our case study is based on the official DOM standard, the compliance test suite provided by the authors of the standard (which is used by browser vendors to show that their browsers faithfully implement the standards), and our own formalization of the standard [3, 4] in Isabelle/HOL [9].

The rest of the paper is structured as follows: in Sect. 2 we present our approach for linking formal and informal parts as well as implementations using test and proof. In the next section (Sect. 3) we report on our experience in applying our approach to the DOM standard [13]. We conclude in Sect. 4.

2 Using Test and Proof for Formalizing Standards

In this section, we present an approach using and combining *test and proof* for providing strong links between semi-formal standards (and their compliance test suites) and a formalization. Fig. 1 illustrates the overall scenario for both traditional development of standards (upper part of the figure) and the integration of "test and proof"-activities (bottom part).

First, let us recall the process and challenges of developing informal or semi-formal standards and implementations that should comply with such a standard: most standards are developed as a text document that contains technical details, e. g., in the form of interface specifications or pseudo-code, that implementations need to comply with. Such semi-formal or informal standards usually contain many inconsistencies; tool support for ensuring the *syntactic consistency* of the

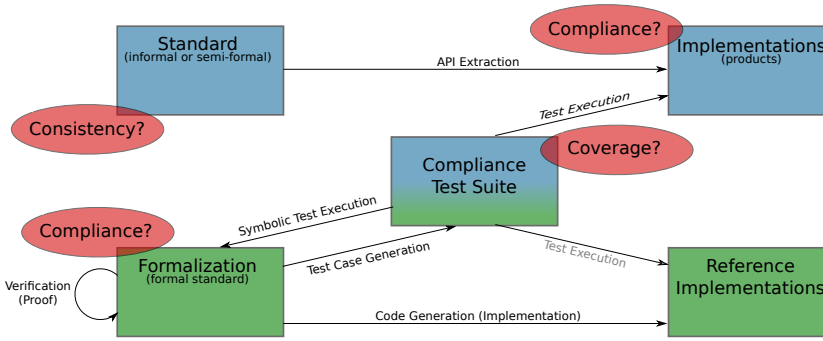

Fig. 1. Using test and proof for establishing strong links between formal standards, compliance test suites, and implementations.

standard is sometimes available in a limited form, but the *semantic consistency* is an open problem. Also, linking standards to implementations is, in the best case, only supported by the possibility to automatically extract interface definitions (APIs), if the standard defines a (software) system. Alternatively, if the standard defines a data format (or a language) it might possible to extract grammar definitions for the abstract or concrete syntax of the defined data format or language. A good standard also includes an extensive set of compliance test cases. These compliance test cases are usually specified manually by experts. Hence, manually developed test cases cannot guarantee to *cover* all important cases and, thus, they can only provide a weak *compliance*-relationship between standard and implementation. Nevertheless, they are the only machine-checkable artifact for vendors to validate the compliance of their product to the standard.

Second, let us discuss how *test and proof* can improve the situation and address the consistency and compliance challenges of semi-formal and informal standard development. In the following, we assume that an executable formalization (e. g., expressed in Isabelle/HOL) of the standard exists. Of course, if we start with an informal standard, the question arises to which extent the formalization is a faithful representation of the informal (or semi-formal) standard, i. e., the *compliance* of the formalization. As we assume an executable model, we can—similarly to implementations—use *symbolic execution* to show the compliance of the formal model to the semi-formal standard (or, more precisely, the manually developed compliance test suite). In addition, we can use the formal model to actually *prove* important properties of the standard (e. g., proving the correctness of the algorithms presented in the standard). We can also generalize test cases provided in the compliance test suite and turn them into proof obligations for our formal model. Using *symbolic specification-based* test case generation techniques (e. g., as presented in [6]), we can automatically generate new compliance test cases that, e. g., guarantee branch coverage on the level of the specification. Finally, we could generate a reference implementation using code generators available in systems such as Isabelle [9] or Coq [2].

3 Case Study: The Document Object Model (DOM)

We successfully applied this approach to our formal model [3, 4] of the DOM standard [13]. This increases the confidence that our formalization faithfully represents the official standard.

3.1 Formalizing the DOM Standard

We illustrate our approach using the `insertBefore` method as an example. The interface of `insertBefore` is given in Web IDL [11]:

```
interface Node {
  Node insertBefore(Node node, Node? child);
}
```

The behavior of this method is described using structural English:

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

This descriptions refers, using hyperlinks, to the concepts *pre-inserting* and *context object*. Without a clear understanding of these concepts, we cannot formalize `insertBefore`. The concept *pre-inserting* is described as follows:

```
pre-insert:
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.
```

Again, several new concepts are introduces and to fully understand the behavior of `insertBefore`, we need to understand and formalize these concepts as well. We formalize the `insertBefore` using monads in Isabelle/HOL:

```
definition insert_before :: "_ object_ptr_Core_DOM ⇒_ node_ptr_Core_DOM
                           ⇒ _ node_ptr_Core_DOM option ⇒_ 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
  }"
```

3.2 Showing Standard Compliance

While our formalization tries to stay as close as possible to the description in the standard, it is not obvious that it complies to it. To show this compliance, we first selected all relevant test cases from the official DOM compliance test suite [12], i. e., the test suite used by web browser vendors to show that their DOM implementation complies to the standard. These test cases are written in JavaScript, which is embedded into the DOM document under test. We then automatically translated these tests into higher-order logic (HOL) to *symbolically execute* (the test cases can be "evaluated" by Isabelle's simplifier using a set of simplifier rules optimized for code generation) them on our model of the DOM. For example, consider the following test case (the left-hand site shows the official specification in JavaScript, the right-hand site our formalization in Isabelle/HOL):

```
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.')
```

```
lemma "test (do {
  a ←document.createElement(''div'');
  b ←document.createElement(''div'');
  c ←document.createElement(''div'');
  assert_throws(NotFoundError,
    (cast a).insertBefore(cast b,
      Some (cast c)))
}) Node_insertBefore_heap"
  by code_simp
(* 'Calling insertBefore with a reference
  child whose parent is not the context
  node must throw a NotFoundError.' *)
```

This test checks whether the DOM method `insertBefore` throws a certain exception if called with a certain combination of arguments. We formalized this test into a state-exception-monad and show the error-freeness by symbolic execution.

Tests are, of course, a very limited way of showing such important properties, as they only show the property for concrete input values (here, a simple DOM instance). To overcome this limitation, we generalize such test cases in to generic theorems that show the corresponding property for all possible inputs. In our example, we generalize the test into the following theorem, which we prove formally in Isabelle/HOL:

```
lemma insert_before_non_child_reference_node:
  assumes "heap_is_wellformed h" and "is_known_ptr_Core_DOM ptr"
    and "¬ (h ⊢reference_child.parentNode →r Some element)"
    and "¬ (is_character_data_ptr element)"
    and "⋀ancestors. h ⊢get_ancestors element →r ancestors
      ⟹cast new_child ∉set ancestors"
shows "h ⊢element.insertBefore(new_child, Some reference_child)
                                    →e NotFoundError"
```

Instead of creating three concrete elements, we can quantify over all possible elements. The two assumptions give additional insight; the test would fail if the argument were a `CharacterData` or included in the reference's ancestors, because these circumstances are checked earlier and cause different exceptions.

Using this approach, we formalized all non-type-related test cases from the official test suite to "test" our model. Table 1 shows the number of formalized

Table 1. The number of tests regarding our supported DOM methods that are available from the official suite and *not* related to type checks. Additionally, we present a rough estimate of the complexity of the tested function along with the coverage of the tests to estimate how much each function would benefit from automatically generated tests.

	# Test Cases in Scope	Function Complexity	Function Coverage
`assignedNodes`	24	high	high
`assignedSlot`	24	high	high
`insertBefore`	5	high	low
`getElementByID`	10	medium	medium
`removeChild`	8	medium	medium
`attachShadow`	2	medium	medium
`createElement`	49	medium	low
`adoptNode`	2	medium	low
`getRoot`	3	medium	low
`childNodes`	2	low	medium
`parentNode`	3	low	medium
`shadowRoot`	2	low	low
`host`	1	low	low
`getOwnerDocument`	0	low	–
`getAttribute`	0	low	–
`setAttribute`	0	low	–
`nextSibling`	0	low	–

tests per DOM function that we support. We cannot easily utilize test cases regarding type checks, as we decided to formalize a strongly typed model. The official compliance test suite contains many typing-related tests, mainly due to two reasons:

1. Dynamic typing and prototype-based inheritance of JavaScript leads to many tests that, for example, check the behavior of functions when passed `null` or `undefined`, whereas we in HOL only allow `None` in places where the DOM standard actually permits it.
2. We model a simplified version of the core DOM. We turned many classes that extend the `Node` interface and, thus, participate in the node tree, into attributes of other interfaces. For example, the DOM standard defines `DocumentType` as a node that must appear in exactly one location of the node tree—it must be the first child of a `Document`. We model the document type as a field of a `Document`. Many tests of the official suite test that constraint, which we therefore did not formalize.

The official test suite is developed manually and, thus, it is not surprising that the test cases vary in style and quality. For example, the compliance test for the tree-modifying method `insertBefore` consists of 26 test cases, of which only five are relevant for our formalization. This indicates that the test authors' concern is mostly testing the absence of run time errors and, to a lesser extent, the correctness of this rather complex method.

3.3 Formal Verification: Analyzing the Standard

In many cases, the methods defined in the DOM standard need to fulfill important properties. These properties are neither spelled out explicitly nor does the compliance test suite contain test cases for them. During the formalization of the standard, these properties often emerge as proof obligations that need to be shown to be able to prove the high-level properties specified in the standard.

An example for such an important property is that after a successful call of `insert_before`, the list of child nodes remains distinct, even if the new child was already a child of that node:

```
lemma insert_before_children_remain_distinct:
  assumes "heap_is_wellformed h" and "is_known_ptr_Core_DOM ptr"
    and "⋀parent. h ⊢get_parent new_child
           →r Some parent ⟹is_known_ptr_Core_DOM parent"
    and "h ⊢insert_before ptr new_child child_opt →h h2"
  shows "⋀ptr children. is_known_ptr_Core_DOM ptr
         ⟹h2 ⊢get_child_nodes ptr →r children
         ⟹distinct children"
```

This is true because `insert_before` first removes the new child from its old parent before inserting it into the child node list of the new parent.

While the verification as such is important to ensure the consistency and implementability of the standard, it also forms the basis for developing an improved compliance test suite. Using a specification-based or theorem prover-based test-case generation approach [6], the proven lemmas can be systematically turned into additional compliance test cases that ensure that actual implementations fulfill these crucial properties.

4 Conclusion

We reported on a first case study combining test and proof for formalizing a standard that is the core of modern web-browsers. We can show the compliance of our formal model to the standard by symbolically executing the official compliance test suite. Our manual analysis of this test suite revealed several important properties that not sufficiently covered, or not covered at all, by the compliance test suite.

As future work, we plan to automatically generated test cases from our formal model (e. g., using HOL-TestGen [5]) and to contribute them to the official compliance test suite. We also plan to enrich our model with a security model formalizing common web-related security measures to verify and test the security guarantees of modern web browsers (and applications running on top of them).

References

[1] Arenis, S.F., Westphal, B., Dietsch, D., Muñiz, M., Andisha, A.S., Podelski, A.: Ready for testing: ensuring conformance to industrial standards through formal

verification. Formal Asp. Comput. **28**(3), 499–527 (2016). http://doi.org/10.1007/s00165-016-0365-3

[2] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag, Heidelberg (2004)

[3] Brucker, A.D., Herzberg, M.: The core DOM. Archive of Formal Proofs (2018). http://www.isa-afp.org/entries/Core_DOM.shtml, Formal proof development. Submitted.

[4] Brucker, A.D., Herzberg, M.: A formal semantics of the core DOM in Isabelle/HOL. In: WWW'18 Companion: The 2018 Web Conference Companion. ACM Press (2018). http://doi.org/10.1145/3184558.3185980.

[5] Brucker, A.D., Wolff, B.: HOL-TestGen: An interactive test-case generation framework. In: Chechik, M., Wirsing, M. (eds.) Fundamental Approaches to Software Engineering, no. 5503 in Lecture Notes in Computer Science, pp. 417–420. Springer-Verlag (2009). http://doi.org/10.1007/978-3-642-00593-0_28.

[6] Brucker, A.D., Wolff, B.: On theorem prover-based testing. Formal Aspects of Computing **25**(5), 683–721 (2013). http://doi.org/10.1007/s00165-012-0222-y.

[7] Horl, J., Aichernig, B.K.: Validating voice communication requirements using lightweight formal methods. IEEE Software **17**(3), 21–27 (2000). http://doi.org/10.1109/52.896246

[8] Kristoffersen, F., Walter, T.: TTCN: towards a formal semantics and validation of test suites. Computer Networks and ISDN Systems **29**(1), 15–47 (1996). http://doi.org/10.1016/S0169-7552(96)00016-5

[9] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer-Verlag, Heidelberg (2002). http://doi.org/10.1007/3-540-45949-9

[10] W3C: W3C DOM4 (2015). https://www.w3.org/TR/dom/

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

[12] W3C: Web platform test: DOM (online. Last visited on 2017-11-10). https://github.com/w3c/web-platform-tests/tree/master/dom

[13] WHATWG: DOM – living standard (2017). https://dom.spec.whatwg.org/commit-snapshots/6253e53af2fbfaa6d25ad09fd54280d8083b2a97/. Last Updated 24 March 2017