The University Of Sheffield.

Formal Foundations for Provably Safe Web Components

By:

Michael Stefan Herzberg

A thesis submitted in partial fulfilment of the requirements for the degree
of
Doctor of Philosophy

The University Of Sheffield
Faculty of Engineering
Department of Computer Science

Submission Date:
11th December, 2019

Abstract

One of the cornerstones of modern software development that enables the creation of sophisticated software systems is the concept of *reusable software components*. Especially the fast-paced and business-driven web ecosystem is in need of a robust and safe way of reusing components. As it stands, however, the concepts and functions needed to create web components are spread out, immature, and not clearly defined, leaving much room for misunderstandings.

To improve the situation, we need to look at the core of web browsers: the Document Object Model (DOM). It represents the state of a website with which users and client-side code (JavaScript) interact. Being in this central position makes the DOM the *most central and critical part* of a web browser with respect to safety and security, so we need to understand exactly what it does and which guarantees it provides. A well-established approach for this kind of highly critical system is to apply formal methods to mathematically prove certain properties.

In this thesis, we provide a formal analysis of web components based on shadow roots, highlight their short-comings by proving them unsafe in many circumstances, and propose suggestions to provably improve their safety. In more detail, we build a formalisation of the Core DOM in Isabelle/HOL into which we introduce shadow roots. Then, we extract novel properties and invariants that improve the often implicit assumptions of the standard. We show that the model complies to the standard by symbolically evaluating all relevant test cases from the official compliance suite successfully on our model. We introduce novel definitions of web components and their safety and classify the most important DOM API accordingly, by which we uncover surprising behavior and shortcomings. Finally, we propose changes to the DOM standard by altering our model and proving that the safety of many DOM API methods improves while leading to a less ambiguous API.

Declaration

I, the author, confirm that the Thesis is my own work. I am aware of the University's Guidance on the Use of Unfair Means (`www.sheffield.ac.uk/ssid/unfair-means`). This work has not been previously been presented for an award at this, or any other, university.

Publications arising from this thesis:

1. Achim D. Brucker and Michael Herzberg. 'A Formal Semantics of the Core DOM in Isabelle/HOL'. in: *Companion Proceedings of the The Web Conference 2018*. WWW 18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 741–749. ISBN: 978-1-4503-5640-4. DOI: 10.1145/3184558.3185980

2. Achim D. Brucker and Michael Herzberg. 'Formalizing (Web) Standards - An Application of Test and Proof'. In: *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*. Ed. by Catherine Dubois and Burkhart Wolff. Vol. 10889. Lecture Notes in Computer Science. Springer, 2018, pp. 159–166. DOI: 10.1007/978-3-319-92994-1_9

3. Achim D. Brucker and Michael Herzberg. 'A Formally Verified Model of Web Components'. In: *Formal Aspects of Component Software - 16th International Conference on Formal Aspects of Component Software, 23-25 October 2019, Amsterdam, Proceedings*. 2019

4. Achim D. Brucker and Michael Herzberg. 'The Core DOM'. in: *Archive of Formal Proofs* (2018). `https://www.isa-afp.org/entries/Core_DOM.html`, Formal proof development. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

5. Achim D. Brucker and Michael Herzberg. 'Shadow DOM: A Formal Model of the Document Object Model with Shadow Roots'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

6. Achim D. Brucker and Michael Herzberg. 'A Formalization of Web Components'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

7. Achim D. Brucker and Michael Herzberg. 'The Safely Composable DOM'. in: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

8. Achim D. Brucker and Michael Herzberg. 'Shadow SC DOM: A Formal Model of the Safelty Composable Document Object Model with Shadow Roots'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

9. Achim D. Brucker and Michael Herzberg. 'A Formalization of Safely Composable Web Components'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

Acknowledgements

I would like to express my sincere gratitude to:

- My supervisor Achim Brucker, who spent countless hours introducing me to Isabelle/HOL and the world of academia, and who never grew tired of just another round of revising drafts.
- My thesis panel, Anthony Simons and Eleni Vasilaki, who gave valuable suggestions throughout my PhD and helped keeping me on track.
- The Isabelle group at the University of Sheffield, who always gave valuable technical and moral support and broadened my understanding of the many uses of the tool.
- The Department of Computer Science of the University of Sheffield, whose scholarship supported me throughout my research and enabled me to travel to conferences.
- All the anonymous reviewers of my conference papers, who gave helpful feedback.
- All my friends and family, who provided all kinds of support and encouragement.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

For many years now, software is everywhere. It runs on every car, keeps planes in the air, assists surgeries, manages investments, and much more. Ensuring that software performs these tasks reliably and correctly is crucial to the safety of many people and has thus been a focus of research for decades. The use of *formal methods* has been one of the proposed solutions, one which couples software engineering with the rigorous methods of mathematical modelling and logic. When applied, they reduce ambiguities of specifications, increase developer productivity by leveraging powerful automated static and dynamic code analysers, and provide mathematical proofs that algorithms have been implemented according to their specification.

However, this rigorous design of software systems is expensive. It can take a lot of time and expertise to formally verify even small software systems. Even though tools supporting such formal developments of software have been around for over 40 years, they have started only recently to see wide-spread use also on larger scale software projects [60] such as the seL4 micro-kernel [42] or a "conference management system with verified document capabilities" [40], both formalised in Isabelle/HOL. Still, the size of modern software systems has also increased, so often a compromise must be reached, concentrating formalisation efforts on the most critical parts.

Therefore, in order to leverage formal methods also for modern, large software systems, it is crucial that they are built modularly, supported by a *robust component system*. Then, one can concentrate their verification and testing efforts on the critical parts of the system while being able to rely on the interface specifications of its dependencies, knowing that they will be enforced by the component system. This kind of approach can alleviate the need for formal verification of the whole system while still providing formal guarantees of security and safety properties. Well specified and loosely coupled components, therefore, can then be analysed in isolation before composing the results to achieve the desired guarantees about the whole system.

One area that has not seen much attention from the formal methods community is the web. We have now reached a point, however, where the web browser has become the most heavily used program on personal computers and mobile phones. People use web browsers for many sensitive tasks such as emails and banking, involving much privacy-critical information. Other information, such as a user's browsing history, which is not necessarily directly related to these activities themselves, can be even more sensitive [38]. Since web browsers have become ever more complex, it is almost impossible to trust them to keep information private and display websites honestly. Therefore, we need to find ways to break down this complexity and provide more guarantees. As web browsers become more

and more similar to operating systems [69], adopting formal method approaches that are already established there seems logical.

The business-driven and fast-paced web ecosystem has seen an increase in the number of developers almost unlike any other field, with many of them being relatively inexperienced. Therefore, a robust and safe way of reusing components created by others is of great value. As it stands, the ability to create components is spread across four different specifications: the ECMAScript Language Specification, which is implemented by, e. g., JavaScript; the Hypertext Markup Language (HTML) standard, which provides a way to write and exchange websites; the Custom Elements standard defining a mechanism for users to introduce new HTML elements; and the *Document Object Model (DOM)* standard. Besides their definitions being spread out, their notion of web components is still rather immature, ambiguous, and has been added to the specifications as an afterthought, leaving much room for misunderstandings. Some of this is due to the history of the official DOM standard being complicated. The Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C), as well as various web browser vendors, have created standardization documents and collaborated to various degrees over the years. Recently, however, those parties have agreed [67] upon one standardization document managed by the WHATWG, which we will refer to as the *official DOM standard* [71]. It is continuously updated ("Living Standard"), which means that we refer to a specific immutable snapshot of this document, which we cite in [71].

In practice, a common way of building and using web components today is the use of certain JavaScript patterns, i. e., attaching all functionality of a module to a single object and then exporting it, along with a package managing system such as npm [53]. This is often coupled with HTML custom elements and templates, which can be used to encapsulate some functionality also on the DOM level. However, such unenforced patterns, together with other properties of JavaScript, are making it impossible to safely include third-party code, especially since they do not separate access to one fundamental data structure that is shared by all JavaScript code: the DOM.

The tree-like structure of the DOM lies at the core of web browsers. It represents the state of a website which users and client-side code (JavaScript) interact with. The part of the DOM that is concerned with web components is *shadow roots*, a mechanism to build a multi-tiered tree that prevents access to certain parts of the tree. Being in between the user and the client-side code makes the DOM the *most central and critical part* of a web browser, so we need to understand exactly what it does and which guarantees it provides.

A well-established approach for this kind of highly critical system is, as we have seen earlier, to apply formal methods and verification to mathematically prove certain properties. One popular tool to aid the formalisation of such software standards is interactive theorem prover Isabelle/HOL [52]. It provides an environment where one can create new types and definitions and prove properties about them using an expressive higher-order logic, while at the same time only having a small set of axioms and code base that the user needs to trust. Therefore, it is an ideal starting point to create a *technical basis for a proof system* for the DOM standard that 1. provides a *consistency guarantee*, 2. enables definitions that are *executable* and can therefore be connected to,

for example, the compliance test suite of web browsers, and 3. remains *extendable* in the sense of [19], so more and more concepts from the core of web browsers can be added with ease.

1.2 Contributions

In this thesis, we present our formal model and analysis of the DOM with shadow roots on which we base a novel definition of web components. Using this formal framework, we then provide proofs of important component separation properties and finally give recommendations for improvements of the standard and possible tools that could aid the safe development of third-party libraries. In more detail, as our contributions, we

1. extract novel definitions of web components and their safety and classify the most important DOM APIs accordingly, by which we uncover surprising behaviour and shortcomings;

2. introduce shadow roots into our model of the DOM as an iterative extension and extract novel properties and invariants that improve the often implicit assumption of the official DOM standard;

3. propose changes to the DOM standard by altering our model and proving that the safety of many DOM API methods improves while leading to a less ambiguous API;

4. show that the model complies to the standard by symbolically evaluating all relevant test cases from the official compliance suite [68] successfully on our model; and finally,

5. build an executable and extendable formalisation of the Core DOM in Isabelle/HOL, based on mathematical logic, that we call *fDOM*, short for *formal Document Object Model*.

This research advances the state-of-the-art in formal development of standard-based software by providing a a significant extension to the DOM standard in a formally verified way, adding a new, safe component mechanism to a major web standard.

The Isabelle theory files on which this thesis is based are all available online in the Archive of Formal Proofs (AFP). We split them into six submissions, as different parts might be useful for different purposes, besides using them to explore the content of this thesis in greater depth: 1. The *Core DOM* [16], which covers Chapter 3 and the Core DOM-related test cases from Chapter 5. This entry is well suited as an entry point for users that are interested learning more about our way of formalising object-oriented standards in general, or are interested in starting with a standard compliant Core DOM formalisation and extending it into a different direction besides shadow roots and components. 2. The *Shadow DOM* [14], which covers Chapter 4 and the Shadow DOM-related test cases from Chapter 5. This entry is well suited as an entry point for users that are interested in exploring shadow roots and their algorithms in more detail.

3. The *Web Components* [11], which covers Chapter 6 and contains everything related to our definition of web components. While the shadow roots-related part remains faithful to the standard, the addition of web components is *not* based upon the standard, but part of our contribution. 4. The *Safely Composable DOM* [17], which covers parts of Chapter 7 and thus contains a version of the DOM that is modified as described. This entry is best compared with the Core DOM entry on a file-by-file basis, highlighting the difference in the data model and the resulting difference in the proofs. 5. The *Safely Composable Shadow DOM* [15], which covers parts of Chapter 7 and contains a version of the Shadow DOM that is built upon our safely composable DOM. This entry is similar to [14] except only necessary changes to the proofs, so this entry is also best compared on a file-by-file basis. 6. The *Safely Composable Web Components* [10], which covers the main part of Chapter 7 and formalises the goal of the modified DOM: a definition of web components that we proof to be safe according to our definition in this chapter.

1.3 Thesis Structure

After introducing Isabelle and higher-order logic (HOL) as well as the DOM and HTML standards briefly in Chapter 2, we introduce the formal data model of the DOM and operations over the DOM in Chapter 3. In Chapter 4, we add shadow roots to our model, serving as the algorithmic basis for our web components. In the following Chapter 5, we leverage the official compliance test suite of the DOM standard to increase the confidence that until this point, we have developed a faithful model of the DOM and shadow roots.

Afterwards in Chapter 6, we depart from the DOM standard to add our own definitions of web components into *fDOM* and prove that the natural approach implied by shadow roots–given a standard-compliant DOM with shadow roots–falls short of our expectations. In order to improve upon this situation, we slightly modify *fDOM* in Chapter 7, update our definitions of web components to more sophisticated ones, and finally prove that those changes indeed improve upon the guarantees of web components–all being backed by the strong guarantees given by Isabelle/HOL. In the end, we will take a look at related work in Chapter 8 and conclude in Chapter 9.

1.4 Externally Reviewed Parts

This thesis contains contributions that have already been presented in form of three conference publications:

1. A conference paper presenting the formalisation of the Core DOM (Chapter 3):

   Achim D. Brucker and Michael Herzberg. 'A Formal Semantics of the Core DOM in Isabelle/HOL'. in: *Companion Proceedings of the The Web Conference 2018*. WWW 18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 741–749. ISBN: 978-1-4503-5640-4. DOI: 10.1145/3184558.3185980

2. A conference paper describing our approach for combining test and proof in the context of software specifications, i. e., combining compliance test suites with symbolic execution (Chapter 5):

   Achim D. Brucker and Michael Herzberg. 'Formalizing (Web) Standards - An Application of Test and Proof'. In: *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*. Ed. by Catherine Dubois and Burkhart Wolff. Vol. 10889. Lecture Notes in Computer Science. Springer, 2018, pp. 159–166. DOI: 10.1007/978-3-319-92994-1_9

3. A conference paper introducing shadow roots into our DOM formalisation and presenting our novel notion of web components (Chapter 4 and Chapter 6):

   Achim D. Brucker and Michael Herzberg. 'A Formally Verified Model of Web Components'. In: *Formal Aspects of Component Software - 16th International Conference on Formal Aspects of Component Software, 23-25 October 2019, Amsterdam, Proceedings*. 2019

In addition, the underlying Isabelle/HOL formalisation has been published in the Archive of Formal Proofs (AFP) [26]:

1. The formal document containing the formalisation of the Core DOM:

   Achim D. Brucker and Michael Herzberg. 'The Core DOM'. in: *Archive of Formal Proofs* (2018). https://www.isa-afp.org/entries/Core_DOM.html, Formal proof development. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

2. The formal document containing the formalisation of the Shadow DOM:

   Achim D. Brucker and Michael Herzberg. 'Shadow DOM: A Formal Model of the Document Object Model with Shadow Roots'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

3. The formal document containing our definition of web components with proofs:

   Achim D. Brucker and Michael Herzberg. 'A Formalization of Web Components'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

4. The formal document containing the formalisation of our altered version of the DOM ("Safely Composable DOM"):

   Achim D. Brucker and Michael Herzberg. 'The Safely Composable DOM'. in: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

5. The formal document containing the formalisation of our altered version of the Shadow DOM ("Safely Composable Shadow DOM"):

   Achim D. Brucker and Michael Herzberg. 'Shadow SC DOM: A Formal Model of the Safelty Composable Document Object Model with Shadow Roots'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

6. The formal document containing the improved definition of DOM components, built on top of the safely composable Shadow DOM:

   Achim D. Brucker and Michael Herzberg. 'A Formalization of Safely Composable Web Components'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x

2 Background

2.1 Isabelle/HOL

2.1.1 Basic Definitions

Isabelle/HOL [52] is an interactive theorem prover implemented in the functional programming language SML and supporting Higher-order Logic (HOL). It supports conservative extensions of definitions, datatypes, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableau provers.

HOL [23, 3] is a classical logic with equality and enriched with total parametric-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 ⇒ 'b`. HOL is centered around the extensional logical equality `_ = _` with type `'a ⇒ 'a ⇒ bool`, where `bool` is the fundamental type of the logic.

The type discipline rules out paradoxes such as Russel's paradox in untyped set theory. Sets of type `'a set` can be defined as isomorphic to functions of type `'a ⇒ bool`; the element-of-relation `_ ∈ _` has the type `'a ⇒ 'a set ⇒ bool` and corresponds basically to the function application. In contrast, the set comprehension `{_ . _}` (usually written {_ | _} in textbooks) has type `'a set ⇒ ('a ⇒ bool) ⇒ 'a set` and corresponds to the λ-abstraction.

Isabelle/HOL supports 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` (`x` of type `'a option`). Moreover, 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 can 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
```

By using the product type (`_ × _`) and the sum type (`_ + _`), it is possible to create type polynoms representing complex types. Since long type polynoms can negatively

affect the readability of definitions and other constructs, we introduce a short-hand notation 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 as an Isabelle theory [16] and thus fully supported by the framework.

In Isabelle, properties are stated using the **lemma** keyword. For example, to express that any arbitrary, but fixed `x :: 'a option` is either `None` or that it contains a value, we can write

```
lemma not_None_eq: "x ≠ None ⟷ (∃y. x = Some y)"
  by (induct x) auto
```

Most notation in Isabelle is purposefully kept close to the mathematical notation that one might find in textbooks.

Now, to prove such a property, Isabelle does not only support simple forward and backward chaining of specific facts, but also more powerful and automated *proof tactics*. For example, `not_None_eq` can be shown by induction on the variable `x` from the lemma over the datatype `option`, followed by the use of the proof tactic `auto` that automatically applies a whole range of previously proven lemmas to the proof goal.

2.1.2 Extending Logics

When extending logics, two approaches can be distinguished: the *axiomatic method* and *conservative extensions*. Extending the HOL core via axioms, i. e., introducing new, *unproven* laws seems to be the easier approach but it easily leads to inconsistency. Given the fact that in any major theorem proving system the core theories and libraries contain several thousand theorems and lemmas, the axiomatic approach is not needed in practice. In contrast, a conservative extension introduces new constants (via *constant definitions*) and types (*type definitions*) only by using a particular schema of axioms; the (meta-level) proof that axioms of this schema preserve consistency can be found in [32]. For example, a constant definition introduces a "fresh" constant symbol and a non-recursive equality axiom with the new constant on the left and a closed expression on the right-hand side. Thus, the new constant can be viewed as an abbreviation of previously defined constructs.

When embedding a language inside a logic, there are again two techniques that can be distinguished: First, using a *deep embedding* of the language into the logic, which means that the abstract syntax is represented as a datatype, which means that we then need to define a semantic function I from syntax to semantics. Second, *shallow embeddings* define the semantics directly; each construct is represented by some function on a semantic domain. Shallow embeddings have the downside, however, that certain characteristics can not easily be referred to, e. g. one could not easily speak about the variables occurring in an expression.

Assume we want to embed a simple language that consists of the two mathematical operators `Add _ _` and `Mul _ _` representing the common addition and multiplication on natural numbers, respectively, into HOL. The semantics then is a function `I :: "'a expr ⇒ 'a env ⇒ nat"` that maps expressions and environments to `nat`, where

environments `'a env = "'a ⇒ nat"` map variables to natural numbers. Using a shallow embedding, we define directly:

```
definition "Add x y ≡ λe. x e + y e"
definition "Mul x y  ≡ λe. x e ∗ y e"
```

Shallow embeddings allow for direct definitions in terms of semantic domains and operations on them. In a deep embedding, we define the syntax as a recursive datatype

```
datatype 'a expr =
  Var 'a
| Add "'a expr" "'a expr"
| Mul "'a expr" "'a expr"
```

and the explicit semantic function `I`

```
primrec I :: "'a expr ⇒ 'a env ⇒ nat"
where
  "I (Var x) = (λe. e x)"
| "I (Add x y) = (λe. I x e + I y e)"
| "I (Mul x y) = (λe. I x e ∗ I y e)"
```

This example reveals the main drawback of deep embeddings: the language is more distant to the underlying meta language HOL, i. e., semantic functions represent obstacles for deduction. However, for conducting certain meta-theoretic analysis, deep-embeddings have advantages. Since we are interested in a concise semantic description of the DOM and efficient proof support and not in meta-theoretic proofs, we chose a shallow embedding for the work presented in this thesis.

Functions in HOL are curried and pure, i.e., they take exactly one argument, return exactly one result, and cannot produce side effects. To simulate functions with more than one argument, we let these functions again return a function. Therefore, when reading curried function definitions, it can be helpful to interpret the chain of function definitions in the following way: the last type definition represents the "return value" of the function, whereas the other types in the chain represent arguments to the function. When modeling stateful functions, such as in our case, we usually define functions that take an argument that represents the state and return an updated version (i.e., a map that contains an additional entry) that represents the state change. In practice, one usually uses syntactic sugar to hide these details.

2.1.3 Monads

A common way to aid the definition of imperative-style definitions in functional programming languages is the use of monads, in particular the *state-exception-monad*, which is often used to hide the state passing from function to function which would otherwise have to be done explicitly. We use the implementation presented in [16]. Here, a function, or *program*, is encoded as a function taking a heap (or, more generally, a mutable state) and returning either an exception, or a result and a new heap:

Table 2.1: Syntax introduced to aid the proofs involving monadic functions. All statements are boolean predicates. We use the turnstile notation to resemble a notion of "given the following state, …".

Function `f`, invoked in heap `h` with argument `x`,	
returns value `y`	`h ⊢ f x →r y`
returns new state `h'`	`h ⊢ f x →h h'`
throws exception `ex`	`h ⊢ f x →e ex`
terminates without an error	`h ⊢ ok f x`

```
datatype ('heap, 'e, 'result) prog =
  Prog (the_prog: "'heap ⇒ 'e + 'result × 'heap")
```

As an example, let us assume we define the following program `h` by first calling program `f` and then calling program `g`:

```
definition f :: "'a ⇒ ('heap, 'e, 'b) prog" where ...

definition g :: "'b ⇒ ('heap, 'e, 'c) prog" where ...

definition h :: "'a ⇒ ('heap, 'e, 'c) prog"
where
  "h x = do {
    y ← f x;
    z ← g y;
    return z
  }"
```

We use the `do`-notation to define the program. The expression `y ← f x;` calls program `f` with argument `x` and binds the result to the local variable `y`, which is of type `'b`. If `f` "throws" an error of type `'e`, `h` immediately returns with the same error, skipping the call of `g`, since only the first error is propagated. If `f` does not throw an error, the execution continues with passing the possibly changed state to the call of `g` (hidden by the monad syntax), along with the argument `y`. The result of the execution is then the result of `g`.

Additionally, we introduce new syntax to ease proofs when monadic functions are involved, which can be found in Table 2.1. This gives us a convenient way to reason about parts of programs while maintaining good proof support and syntax that is similar to the definition of the functions.

2.1.4 Locales

Locales [4] are one of Isabelle's mechanism to enable the writing of parametrised theories. To get an idea of how this works, let us consider a small example first. Assume we want to capture the fact that if we assume a predicate `P :: "int ⇒ bool"` which is true for argument `0` and which remains true for all successive values once it has been true, then we also know that the predicate is certainly true for all values greater or equal than `0`,

especially for the value 42. Now if we want to derive these two properties for multiple predicates, we do not want to repeat these proofs for every concrete P. Therefore, we define the following locale which captures this proof once for a general predicate P:

```
locale l_P =
  fixes P :: "int ⇒ bool"
  assumes P_zero: "P 0"
  assumes P_remains_true: "P n ⟹ P (n + 1)"
begin
lemma P_greater_equals_zero_true: "n ≥ 0 ⟹ P n"
  using P_zero P_remains_true int_ge_induct by blast
lemma P_fourty_two: "P 42"
  using P_greater_equals_zero_true by simp
end
```

We prefix the name of the locale with l_ in order to more easily distinguish the locale from the predicate P. As we can see, the lemma P_greater_equals_zero_true follows from our two assumptions and int_ge_induct, a lemma which enables the automatic application of simple induction proofs. By using this lemma, property P_fourty_two is also proven in a straight-forward manner.

Locales provide their own context, opened with **begin** and closed with **end**, inside which one can make use of the specified types and assumptions. This effectively introduces new axioms, but only inside the environment of this locale.

In order to make use of our new lemmas also on the theory level outside of the locale, we need to interpret the locale with a concrete P and show the made assumptions:

```
interpretation P_minus_six: l_P "λn. −6 < n"
proof
  show P_zero: "(− 6::int) < 0"
    by simp
next
  show P_remains_true: "⋀n::int. − 6 < n ⟹ − 6 < n + 1"
    by linarith
qed
```

Here, we use the predicate that evaluates to true for all values greater than minus six. The **interpretation** keyword leaves us with the locale assumptions as proof obligations, which in this case, are easily proven. As a result, we get the lemmas from the locale interpretation on the theory level:

```
thm P_minus_six.P_fourty_two  (* Output: − 6 < 42 *)
```

As we can see, with locales we can prove properties over functions in an abstract way without the need to refer to the actual definition of it. This is especially useful if we have multiple functions that differ only slightly from each other, so we leverage their common properties to prove lemma only once, but gain them for every concrete definition by simply interpreting them.

Besides "generating" lemmas, we can also use this mechanism to easily create new definitions that are similar to each other. This works in the same way as for lemmas.

Another feature of locales that will be important for us later is the extension of locales with other locales. We could now, for example, create a new locale and import our example locale `l_P`. By doing so we gain access to its assumptions, proven lemmas, and defined functions:

```
locale l_Q = l_P +
  fixes ...
```

When using locales in this way, we can create a compositional hierarchy of lemmas and function definitions that can reduce duplication and make the theory files easier to read and maintain.

2.1.5 Code Generator

Isabelle/HOL features an extensive code generator setup which is able to convert datatype and function definitions into SML, Haskell, Scala, and other (functional) programming languages [34]. Not all definitions are suitable, however; since Isabelle supports the full expressivity of higher-order logic, not all definitions can be converted into executable programs. Since we want to make use of the code generator at various points, we will ensure that most of our definitions will be suitable, or will at least provide equivalent substitutions.

In addition to generating code for different languages to be used outside of Isabelle/HOL, there also exist Isabelle proof tactics that can make use of executable definitions. For us, these are mainly the following two:

Using `code_simp` instructs the simplifier to only use code equations. These are usually equations which define new functions, but also equations that, for example, convert a set comprehension into a filtering of all values of a given type. This tactic does not generate any actual code, but works similar to *symbolic execution*, which means it is comparatively slow. Still, as it only uses the simplifier, we do not introduce any complex additional dependencies that would need to be trusted in addition to the Isabelle/HOL core.

With `eval`, Isabelle/HOL internally generates and executes SML code from the current proof goal. It is therefore faster than the simplifier, but has the additional "trust burden" of the code translator, because there might be errors in the implementation of the SML infrastructure.

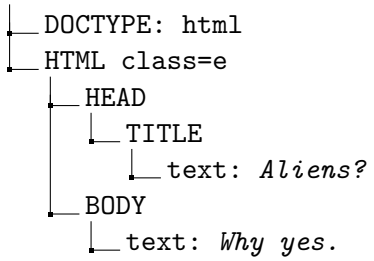
We will use this kind of proof tactic to prove "test cases" for our DOM API specifications, i. e. lemmas that involve concrete DOM heaps instead of arbitrary, but fixed ones. In those cases, we can calculate the concrete heap states after each method by using `code_simp`. For prototyping proofs, we use `eval` instead to achieve faster execution times, but change all occurrences to `code_simp` afterwards to achieve a higher trust level.

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

(a) HTML

Why yes.

(b) Rendering



(c) DOM

Figure 2.1: A simple example of a DOM instance: (a) shows a textual representation using HTML syntax, (b) a visualization of the node-tree of the DOM, and (c) shows the result of rendering this DOM, e.g., by a rendering engine of a web browser.

2.2 Web Standards

2.2.1 Document Object Model (DOM)

At its core, the *Document Object Model (DOM)* defines a tree-like data structure for representing documents in general and HTML documents in particular. Figure 2.1 illustrates a small example: Figure 2.1a shows the textual representation of a simple document (using HTML as syntax), Figure 2.1c shows the visualization of the DOM *node-tree*, and Figure 2.1b shows the rendered output (e. g., in a web browser).

The history of the official DOM standard is complicated; the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C), as well as various web browser vendors, have created standardization documents and collaborated to various degrees over the years. For us only the fact that those parties have since agreed [67] upon one standardization document managed by the WHATWG is relevant, which we will refer to as the *official DOM standard* [71]. It is continuously updated ("Living Standard"), which means that we refer to a specific immutable snapshot of this document, which we cite in [71].

The full standard document is very large, however, and contains many related features and algorithms, not all of which are strictly necessary to specify a simple document. Since we cannot hope to formalise such a big software standard in its entirety, we need to extract and focus on the core of the DOM. We will refer to those parts that we deem relevant for this purpose as the *core DOM*; this must not be mistaken for the "DOM Core Level 1" [22] and successive documents, which is the way how the standard has been developed previously to the adoption of the "Living Standard".

It is important to note that the standard does not limit the application domain of the DOM to web browsers–besides HTML, the DOM also provides the basis for XML. Similarly, the standard only specifies an API and is therefore not limited to using JavaScript as an implementation language, even though it certainly is among the most wide-spread ones.

2.2.2 Hypertext Markup Language (HTML) and JavaScript

The *Hypertext Markup Language (HTML)* is the standard mark-up language for the modern web, and is also standardized by the WHATWG. For us, it is important to keep HTML separate from DOM, since HTML only describes the "serialization format" of the DOM. We focus on the behavior and API defined by the DOM standard. Nevertheless, DOM and HTML are closely linked, as we will see in Chapter 5 that the official DOM test suite uses HTML and JavaScript to test the DOM implementation inside web browsers.

Summary

In this section, we have learned about Isabelle/HOL, the interactive theorem prover upon which we base our DOM formalisation, and therefore we rely quite heavily on it. While we gave a short introduction into the specific features and notations of Isabelle that we use, some familiarity with Isabelle or at least similar tools such as Coq is definitely recommended to fully understand the following chapters. Also, we introduced the Document Object Model (DOM), which is standardised by a standardisation authority and a cornerstone of the web.

3 A Formal Core DOM: *fDOM*

In this chapter, we will introduce our *formal core DOM*: *fDOM*. It is a version of the Document Object Model (DOM) standard that has been reduced to its most essential parts: the node tree. We will present our approach on how to formalise such a software standard in Isabelle/HOL and describe our design decisions. While all claims regarding the DOM have been fully proven within Isabelle/HOL (the theory files can be found online [16]), we will only show excerpts in this thesis, since the full formalisation document is multiple hundred pages long. In this chapter, therefore, we will gain new insights into the DOM itself from a formal methods perspective; its contributions are three-fold:

Firstly, as we build our *fDOM*, we will gain new insights into the DOM standard as such. Being a popular standard that is multiple decades old, we can not reasonably expect to unearth serious inconsistencies; our insights will have a clarifying character, which we consider to be of great value, too, as the official DOM standard leaves many things implicit. For example, we will mine a set of invariants that tree-modifying methods need to satisfy in order to leave the node tree in a "proper" state.

Secondly, the *fDOM* will enable us to verify the functional correctness of common functions that manipulate the node tree, e. g., `insertBefore`. In addition, we will prove that these functions actually preserve these invariants and see which seemingly innocuous conditions of the DOM API are responsible for that.

Thirdly, we will see how the *fDOM* will provide a formal foundation for further analysis of higher-level properties and additions to the standard. For this purpose, we will see features of the *fDOM*, such as its *extensibility* (extending without the need of re-proving already proven properties) and *executability* (generating executable code from our specification), which we will introduce abstractly in this chapter, but it will not be until Chapter 4 and Chapter 5, respectively, that we will see these features in action.

On a technical level, the extensibility is achieved mainly by using name mangling and Isabelle's locale mechanism. Following the idea from [19], we iteratively build our formalisation document by extending our universe one theory file at a time, although not all extensions are equal. For example, we will have multiple extensions for our datatypes (one datatype per extension), before we will have an extension introducing new methods. The following sections will introduce this mechanism in more detail and with examples, although we won't see the real power of this way of extending until we introduce shadow roots in Chapter 4.

3.1 The Core DOM Data Model

Our goal is to model the core part of the DOM: the node tree. Normally, one would model a tree in HOL in a similar way as in a functional programming language; storing

```
datatype
  object = Object and
  node = Node and
  element = Element (tagName: string)
    (childNodes: "(element + character_data) list")
    (attributes: "string ⇀ string") and
  character_data = CharacterData (data: string) and
  document = Document (doctype: string) (documentElement: "element option")
    (disconnectedNodes: "(element + character_data) list")

definition "example = Document ''html'' (
  Some (Element ''html'' [
    Inl (Element ''head'' [
      Inl (Element ''title'' [
        Inr (CharacterData ''Aliens?'')
      ] Map.empty)
    ] Map.empty),
    Inl (Element ''body'' [
      Inr (CharacterData ''Why yes.'')
    ] Map.empty)
  ] (map_of [(''class'', ''e'')]))
) []"
```

Figure 3.1: An attempt to represent the Core DOM basic model using a purely functional tree encoding, using the example from Figure 2.1a. Besides lacking many other features such as type inheritance, it is unclear how a function such as `insert_before` would be implemented, given that it would need the ability to refer to an arbitrary node in the tree in a unique manner.

the values directly in the datatypes. An example of how such a formalisation *could* look like can be found in Figure 3.1. It could have a separate datatype for each kind of node that we have, such as `element` and `character_data`, and instead of *references* to other nodes, we store them directly in the datatype itself, e. g., `element` contains a `childNodes` attribute of type `(element + character_data) list`, instead of a separate pointer datatype or even a simple numbered reference.

For our purposes, however, this standard approach is not suitable. We would like to stay as close to the official DOM standard as possible, which specifies the node tree as an *object-oriented* data structure, using *typed pointers* and objects that are stored inside a heap. In addition, with the purely functional approach, we would be unable to distinguish identical sub-trees, which would make it impossible to model many DOM API functions that use node references to, e. g., determine which node should get deleted. One could add identifiers to the datatypes that would help to distinguish identical sub-trees, but this would require additional constraints that are foreign to the DOM standard and move

the data model further away from it. Also, features like type inheritance are difficult to implement in this way.

Instead, we add an additional argument to all functions that we use to pass a *heap of node trees*, which is a finite set of pairs of pointers and objects and represents the state of the DOM. In our data model and DOM API methods, we will then use pointers that can be resolved using this heap. It must be noticed, however, that this design decision comes at a cost: Compared to the purely functional approach, we will need to introduce additional constraints in order to keep our trees well-formed, e. g., we need to make sure that the pointered structures do not contain cycles. In addition, many proofs involving functions that recurse over the tree will become more complicated as we will need to ensure that the functions terminate. Still, while this way of modelling the state increases the complexity of proofs, the DOM, as specified in the standard, shares many of these drawbacks, so we will later see that the additional effort results in more useful properties.

Note that formalising pointer-based data structures in higher-order logic within Isabelle/HOL has been done before by, e. g., Mehta et. al [49], where "heaps are modelled as mappings from addresses to values, and pointer structures are mapped to higher-level data types". One of the key design decisions of our DOM formalisation, however, is using a typed pointer *hierarchy* including sub-typing and polymorphism to model the DOM, staying close to the official specification. The second key difference between our work and similar ones, such as Tuch et. al [64] combining "Types, Bytes, and Separation Logic" into a separation logic-based framework implemented inside Isabelle/HOL, is the fact that we do not need to handle low-level details of a C-style memory layout, but are able to use a much higher-level abstraction where a heap stores typed pointers that are numbered separately from pointers of other types. It is also important to keep in mind that our goal is to research how suitable our chosen way of formalising the DOM is and how, for example, the frame conditions could look like, rather than to provide an efficient DOM or even JavaScript verification tool.

3.1.1 Classes as Datatypes

Before we can formalise the API, we need to find a suitable formalisation of the underlying data model. As the DOM models a tree-like data structure, it is natural that the core datatypes of the DOM specifications are `Document` and the datatype `Node` with the two specializations `Element` and `CharacterData`. Figure 3.2 shows the set of core datatypes of *fDOM* using Web IDL [44], the notation used by the WHATWG. In our data model, we omit attributes that can be computed from others, e.g., the `parent` attribute which represents the inverse relation already represented by the `childNodes` and `documentElement` attributes, so that we do not need to specify a lot of additional well-formedness constraints. For the same reason we, instead of storing the `ownerDocument` of a `Node`, we store the list of disconnected nodes in `disconnectedNodes`. Figure 3.3 visualises the type hierarchy of the types that we use.

In Isabelle/HOL, *records* are the natural way of representing such hierarchical types with associated attributes. They are similar to ordinary datatypes, but additionally provide a convenient syntax for accessing and updating individual attributes and can

```
interface Object {};
interface Node : Object {};
interface Element : Node {
  readonly attribute DOMString tagName;
  readonly attribute NodeList childNodes;
  readonly attribute NamedNodeMap attributes;
};
interface CharacterData : Node {
  attribute DOMString data;
};
interface Document : Object {
  readonly attribute DocumentType? doctype;
  readonly attribute Element? documentElement;
  readonly attribute NodeList disconnectedNodes;
};
```

Figure 3.2: The IDL specification of the data model of the core DOM. We omit attributes that can be computed from others, such as `parent`. Instead of the `ownerDocument` of nodes, we store the list of disconnected nodes in `disconnectedNodes`. The question mark indicates a "nullable" type. We introduced `Object` to have a common super-interface.

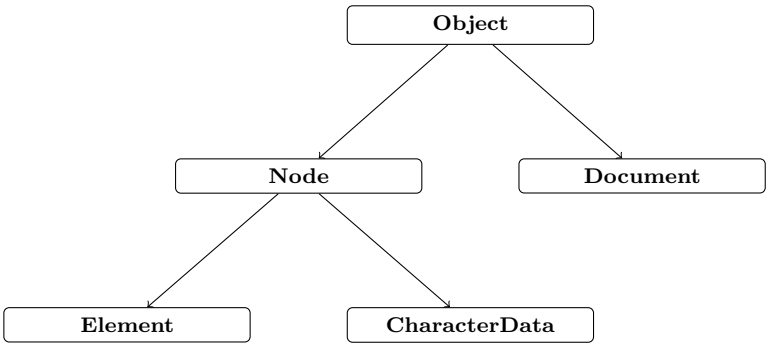


Figure 3.3: The type hierarchy of our DOM data model.

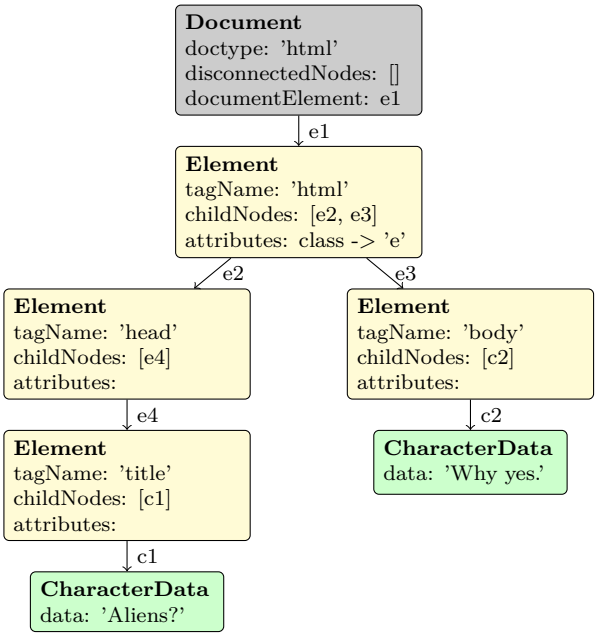


Figure 3.4: Schematic tree-representation of Figure 2.1a that demonstrate the use of the interfaces and attribute of Figure 3.2.

extend other records, which means that they inherit all of their attributes. This makes them ideally suited for our purposes, as these properties keep our definitions that might exhibit different behaviour depending on the actual class more readable. The main benefit though is that records keep the attributes organised in a hierarchical way, much like the DOM specification.

Therefore, we define one record for each class:

```
type_synonym DOMString = string
type_synonym doctype = DOMString
type_synonym attribute_key = DOMString
type_synonym attribute_value = DOMString
type_synonym attributes_type = "(attribute_key, attribute_value) fmap"
type_synonym tag_type = DOMString

record RObject =
  nothing :: unit
record RNode = RObject +
  nothing :: unit
record _ RElement = RNode +
  nothing :: unit
  tag_name :: tag_type
  child_nodes :: "_ node_ptr list"
  attributes :: attributes_type
```

```
record RCharacterData = RNode +
  nothing :: unit
  data :: DOMString
record _ RDocument = RObject +
  nothing :: unit
  doctype :: doctype
  document_element :: "_ element_ptr option"
  disconnected_nodes :: "_ node_ptr list"
```

We add the prefix `R` to distinguish each type from its corresponding type synonym later. Due to technical constraints of the record package, we need to introduce an attribute `nothing` for classes that do not define at least one attribute themselves. We also define some useful type abbreviations for some of the types of the record fields to closely model our WebIDL specification in Figure 3.2. `RObject` acts as our base record type from which all other records inherit.

Given these definitions, we can, e.g., define a concrete `CharacterData` object as follows, using the record syntax to create a new instance:

```
definition "CharacterDataExample =
  ⦇RObject.nothing = (),
   RNode.nothing = (),
   RCharacterData.nothing = (),
   data = ''Why yes.'',
   ... = ()
  ⦈"
```

As we can see, each `RCharacterData` also contains the fields from `RNode` and `RObject`, which, in this case, are only the `nothing` fields, though we will have a class later that makes use of this mechanism. The last assignment, `... = ()`, assigns the unit value to the extension slot of `RCharacterData`, which basically closes the slot and forbids any further extensions. This is part of the extension mechanism provided by records, which allows us to define HOL functions that take an argument of type `RNode`, which we can fill with either `RElement` or `RCharacterData` — a *core mechanism* known from object-oriented programming. As it stands, support of the record package for this is limited, though, as we have access only to fields from `RNode` and `RObject` in this case, but not to either `RElement` or `RCharacterData`!

Thankfully, we can improve upon this shortcoming. By default, the record package hides the type variable of the extension slot, so if we specify a function argument type of `RNode`, we do not have any type information about possible extensions. We can, however, encode our type hierarchy into this extension type variable and later create new type synonyms for them. With the usage of sum types, we can determine the concrete class of the instance at *runtime* and handle these cases differently. In addition, we also use the option type in this construction to distinguish "abstract" classes whose concrete instances can not exist on their own, but will always need to have another, more concrete class. For example, we can define a HOL function that returns `True` if and only if a given `RNode` is actually a `RElement`:

```
definition is_element_kind ::
  "((_, 'Element option) RElement_ext + 'Node) RNode_scheme ⇒ bool"
where
  "is_element_kind node =
    (case RNode.more node of
      Inl element ⇒ True |
      Inr _ ⇒ False)"
```

Isabelle's record package creates for each record an additional datatype with the suffix `_ext` and `_scheme`, which refer to only the extension slot and to the original datatype, but with the possibility to specify the desired type of the extension as a type argument. More information regarding the inner workings of records can be found in the Isabelle documentation [70]. By specifying the type sum in the type argument to `RNode_scheme`, we force the extension slot of node to either be an element, or a yet-to-be-specified `'Node`. In both cases, the node class will be "abstract", i. e., will always have an extension. In contrast, our element type has `'Element option` in its extension slot, so it could either have another extension `'Element`, or none at all. The construction that enhances the record extensibility with support for runtime "down-casting" is rather technical and requires a number of type synonyms not shown here, so we refer the interested reader to the full Isabelle proof document for further details.

In summary, this essentially models 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 class `Element` can be internal nodes or leaves, and 3. instances of the class `CharacterData` can only appear as leaves. As an added benefit, we can use our record construction later to easily *extend our type universe* with more types, e. g., `ShadowRoot`, without having to modify any existing theories or redefine all of our functions. Functions that use our assembled record types using the `_ext` and `_scheme` variants can be written in a more generic way and can contain a fallback case that handles types that are not yet known.

However, before we can actually represent node trees, we not only need objects, but also pointers and a heap to store the mappings from pointers to objects. When we define the DOM methods later, we will only deal with typed pointers and the heap directly, but not with the object types. This allows us to hide much of the complexity of the type hierarchy encoding.

3.1.2 Pointer datatypes

Now, we continue by defining the datatypes required for our pointers, the second part of the datatypes that are necessary to represent a reference-based graph structure. In essence, we try to mimic the structure and inheritance mechanism introduced for the object records, with the main difference being that we do not need fields except for a single natural number that will act as an "address" within our heap. Since we use a finite map as our heap, the only purpose of this address is to distinguish different pointers from each other, because using HOL equality, two objects are equal if all their fields are equal. The location that a pointer references never changes, and we never actually need

to retrieve it, so we do not need the additional complexity of the record package. We will therefore use the more basic **datatype** for our pointer type definitions:

```
type_synonym ref = nat

datatype 'object_ptr object_ptr = Ext 'object_ptr
datatype 'node_ptr node_ptr = Ext 'node_ptr
datatype 'element_ptr element_ptr = Ref ref | Ext 'element_ptr
datatype 'character_data_ptr character_data_ptr = Ref ref
                                                | Ext 'character_data_ptr
datatype 'document_ptr document_ptr = Ref ref | Ext 'document_ptr
```

Again, we see that `object_ptr` and `node_ptr` are visibly different from the other definitions; they only have one type constructor, `Ext`, which takes a single value of the type of the type parameter. On the other hand, `element_ptr`, for example, also has the `Ref` constructor that takes a natural number and thus creates finally a "concrete" pointer to an object. The actual type hierarchy is not encoded in these datatype definitions yet.

We use these datatypes to introduce type synonyms representing the actual pointer types for our DOM model inside the *current type universe*. The actual type hierarchy gets encoded into the type variable polynomials. For example, when adding `element_ptr`, `'node_ptr` gets replaced by `'element_ptr element_ptr + 'node_ptr`:

```
datatype 'element_ptr element_ptr = Ref (the_ref: ref) | Ext 'element_ptr
type_synonym ('node_ptr, 'element_ptr) node_ptr
  = "('element_ptr element_ptr + 'node_ptr) node_ptr"
type_synonym ('object_ptr, 'node_ptr, 'element_ptr) object_ptr
  = "('object_ptr, 'element_ptr element_ptr + 'node_ptr) object_ptr"
```

At this point, only the `Object` and `Node` pointer types exist yet. After the initial declaration of the `element_ptr` type, we continue to overload the `node_ptr` and `object_ptr` types with synonyms according to our construction. The type hierarchy gets encoded by constructing the type polynomials accordingly.

The type polynomials are constructed in such a way that the HOL types for pointers of sub-classes in the object-oriented model are instances of the HOL type of their super-class. This is key to an *extensible* formalisation. By re-defining all existing datatypes and type synonyms for each universe extension, every name always refers to the most up-to-date version.

From here on, we will use an underscore to denote the tuple of type variables of the type constructors for pointer and object types. For example, we will write `_ node_ptr` instead of `('node_ptr, 'element_ptr, 'character_data_ptr) node_ptr` and assume that type variables of the same name are instantiated with the same types.

Finally, we will tie together our pointer and object types and fix the type hierarchy by defining a heap for storing node-trees, i.e., instances of our DOM data model, that we will use for all further definitions. We will usually take a pointer of any type, up-cast it to an `object_ptr` by using the appropriate type constructors (note that pointer reference itself is stored in the most concrete type), resolve it by using the heap, take the retrieved

```
definition aliens_heap :: heap_final where
  "aliens_heap = create_heap [
    (cast (document_ptr.Ref 1),
      cast (create_document_obj html (Some (cast (element_ptr.Ref 1))) [])),
    (cast (element_ptr.Ref 1),
      cast (create_element_obj ''html'' [cast (element_ptr.Ref 2),
        cast (element_ptr.Ref 4)] (fmap_of_list [(''class'', ''e'')]) None)),
    (cast (element_ptr.Ref 2),
      cast (create_element_obj ''head'' [cast (element_ptr.Ref 3)]
        fmempty None)),
    (cast (element_ptr.Ref 3),
      cast (create_element_obj ''title'' [cast (character_data_ptr.Ref 1)]
        fmempty None)),
    (cast (character_data_ptr.Ref 1),
      cast (create_character_data_obj ''Aliens%3F'')),
    (cast (element_ptr.Ref 4),
      cast (create_element_obj ''body'' [cast (character_data_ptr.Ref 2)]
        fmempty None)),
    (cast (character_data_ptr.Ref 2),
      cast (create_character_data_obj ''Why%20yes.''))]"
```

Figure 3.5: The formal representation of a heap containing our simple example DOM (recall Figure 2.1). We use the overloaded function `cast` to convert pointer and object types into other types up and down the type hierarchy. The finite map is then created from the shown list of tuples of pointers and objects.

object and access whichever field we are interested in. Therefore, we define a DOM heap as is a finite map from object pointers to objects:

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

type_synonym heap_final = "(unit, unit) heap"
```

Here, `_ Object` is the type synonym for the instantiated super-type of object (similar to the construction for pointers). In addition to `heap`, we also introduce the shorthand `heap_final`, which refers to a closed heap that does not allow any further type extensions. We will use such heaps to work with concrete DOM instances.

Figure 3.5 illustrates how the simple document from our example in Figure 2.1 can be expressed in our formal DOM heap. We use `adhoc_overloading` to overload a function `cast` to convert pointer and object types into other types up and down the type hierarchy. In most cases, the type inference engine of Isabelle is able to calculate the correct cast function without user guidance. The `create_` functions provide a short-hand to create new instances of our objects by mapping the first argument to the first record field, and so on.

Another important concept regarding the extensibility of our type universe is the notion of a *known pointer*. We consider a concrete instance of a pointer *known* if it does not contain any reference to a free type variable. We will try to keep the types of

our definitions as generic as possible so that they can also be applied on a heap that potentially contains pointer and object types that were not defined yet when the method was defined. For example, for the definition of a getter function retrieving a certain attribute from an element it is not relevant which other, more concrete types of document there might be in the heap. Using our generic heap type, we do not need to redefine such function definitions just to be type-compatible with the new heap. However, there are also functions whose behavior depends on the concrete type at runtime and might not have a default behavior. In these cases, we often want to conduct proofs assuming that we do not have "undefined behavior". For this purpose, we define a predicate `known_ptr` for each extension of our type universe:

```
definition known_ptr :: "(_) object_ptr ⇒ bool"
where
  "known_ptr ptr ⟷ CharacterDataClass.known_ptr ptr ∨ is_document_ptr ptr"
```

This particular predicate is the version of the most recent extension, the one in which `RDocument` was added. It therefore asserts that the given `ptr` is either a `document_ptr` (and *not* a possible extension of it) or a known pointer in one of the previous extensions. In a similar way, we "re-define" a number of different definitions to keep them up-to-update with the most recent extension of our type universe.

Summary

In this section, we have described a construction that maps *object-oriented sub-typing and inheritance* to polymorphic HOL types. To achieve that, we have defined our datatypes for pointers, objects, and the heap that stores mappings from pointers to objects. Together, this provides us with the basis to store our node-tree—we can now store reference-based graphs in HOL. The types that we used are flexible enough so that we can easily extend our type universe later with new types, without having to redefine all of our functions. We can define them once in a generic way so that they can also take types from later type universe extensions. However, if we want to add new functionality based on these new types, we will have to define a new function for that, but we will still be able to reuse previous functionality.

3.2 Defining Operations and Queries on Node-Trees

In the following we will define the DOM methods for creating, querying, and modifying the node-trees that are stored in a DOM heap. We define the following functions formally in Isabelle/HOL. Figure 3.6 provides an overview of their formal type signatures.

All operations use the state-exception-monad introduced in Section 2.1.3. This gives us a convenient and visually similar way to model DOM operations and programs involving them since they are typically written in an imperative style and use exceptions. This similarity helps us to stay as close as possible to the DOM standard. We use our newly defined type `heap` as our heap datatype and introduce a simple datatype `exception` that represents all different exceptions that the DOM API uses:

```
create_element    :: "tag_type ⇒ _ document_ptr ⇒ _ dom_prog"
get_attribute     :: "_ element_ptr ⇒ attributes_key ⇒ _ dom_prog"
set_attribute     :: "_ element_ptr ⇒ attributes_key
  ⇒ attributes_value option ⇒ _ dom_prog"
get_child_nodes   :: "_ object_ptr ⇒ _ dom_prog"
get_parent        :: "_ node_ptr ⇒ _ dom_prog"
remove_child      :: "_ object_ptr ⇒ _ node_ptr ⇒ _ dom_prog"
get_element_by_id :: "_ object_ptr ⇒ attributes_value ⇒ _ dom_prog"
adopt_node        :: "_ document_ptr ⇒ _ node_ptr ⇒ _ dom_prog"
insert_before     :: "_ object_ptr ⇒ _ node_ptr ⇒ _ node_ptr option
  ⇒ _ dom_prog"
```

Figure 3.6: The formal type signatures of the methods for creating, querying, and modifying the core DOM.

```
datatype exception = NotFoundError | HierarchyRequestError | NotSupportedError
  | SegmentationFault | AssertException | NonTerminationException
  | InvokeError | TypeError
type_synonym (_, 'result) dom_prog = "((_) heap, exception, 'result) prog"
```

There are many reasons why a `dom_prog` might throw an exception: `NotFoundError`, `HierarchyRequestError` and `NotSupportedError` are described in the DOM standard and are thrown on errors, e. g. when an operation would violate one of the well-formedness constraints. We throw a `SegmentationFault` whenever a caller tries to dereference a pointer that does not exist in the current heap, `AssertException` is used to model test cases, we use `NonTerminationExceptions` to model potentially non-terminating functions in HOL, which requires all functions to terminate, `InvokeErrors` for our method overloading mechanism, and finally, `InvokeError` whenever we encounter a pointer-object-pair in the heap which types do not match (e. g., an element pointer that does not point to an element). Generally, however, we are more interested in the fact that no errors occur, rather then which specific exception gets thrown.

Now, we illustrate our approach of formalising a DOM method from the standard using the `insertBefore` method as an example. The standard contains a mix of structural English describing the behaviour of the DOM methods and the formal type signatures using Web IDL. The interface of `insertBefore` is given in Web IDL:

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

The behavior of this method is described in the standard 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 formalise `insertBefore`. Therefore, we continue with the concept of *pre-inserting*, which is described in a way that closely resembles pseudocode. The excerpt describing the pre-inserting algorithm looks 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 introduced. To fully understand the behaviour of `insertBefore`, we need to understand and formalise these concepts as well. After considering which ones are important for the core DOM, we formalise the `insertBefore` as follows:

```
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;
    disc_nodes ← get_disconnected_nodes owner_document;
    set_disconnected_nodes owner_document (remove1 node disc_nodes);
    insert_node ptr node reference_child
  }"
```

A node that should be inserted needs to fulfil certain well-formedness criteria. This is checked using the `ensure_preinsertion_validity` function which formalises the concept of *pre-insertion validity* from the DOM standard (Figure 3.7). Then, the "reference child" needs to be determined, which is the node before which the to-be-inserted node should be placed. Then, we *adopt* the node into the (possibly new) node-tree and finally insert the node into either the `child_nodes` or `document_element` attributes. Figure 3.8 shows the effect of `insertBefore` on a small example.

We apply this approach of formalising DOM methods to any method that is related to the node tree. In the following, we will summarize the most common methods to give a better idea about their type signatures and their behavior.

The function `create_element` takes an (owner)document and the tag name of the new element. It returns the updated heap that includes the new element with no children

```
To ensure pre-insertion validity of a node into a parent before a child,
    run these steps:
  1. If parent is not a Document, DocumentFragment, or Element node,
    throw a "HierarchyRequestError" DOMException.
  2. If node is a host-including inclusive ancestor of parent, throw
    a "HierarchyRequestError" DOMException.
  3. If child is not null and its parent is not parent, then throw a
    "NotFoundError" DOMException.
  4. If node is not a DocumentFragment, DocumentType, Element, Text,
    ProcessingInstruction, or Comment node, throw a
    "HierarchyRequestError" DOMException.
  5. If either node is a Text node and parent is a document, or node
    is a doctype and parent is not a document, throw a
    HierarchyRequestError" DOMException.
  6. If parent is a document, and any of the statements below,
    switched on node, are true, throw a "HierarchyRequestError"
    DOMException.

   DocumentFragment node
     If node has more than one element child or has a Text node child.

     Otherwise, if node has one element child and either parent has
       an element child, child is a doctype, or child is not null and
       a doctype is following child.
   element
     parent has an element child, child is a doctype, or child is not
     null and a doctype is following child.
   doctype
     parent has a doctype child, child is non-null and an element is
     preceding child, or child is null and parent has an element child.
```

Figure 3.7: The semi-formal specification of the concept “pre-insertion validity” from the official standard [71].

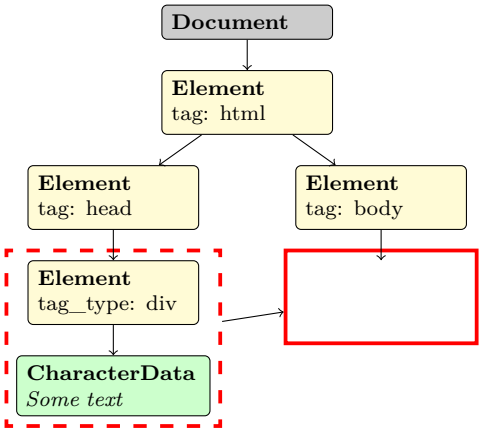


Figure 3.8: An example tree illustrating the possible effects of calling `insert_before`. If the content of the red dashed box gets inserted below the `body` element, the whole sub-tree gets moved to the new location (the red solid box).

and no attributes along with a reference to the new element, which is stored in the first free location in the heap. This ensures that it will not change any existing locations in the heap, which we will prove later. Additionally, the new element is added to the list of disconnected nodes of the given document, as it is not yet part of the node tree.

The function `get_child_nodes` takes a heap and a pointer to a node and returns a list of pointers to its children. For elements, it returns the children list that is stored in the datatype. For text nodes, it returns the empty list. For documents, we convert their document element into the appropriate node list.

The function `get_attribute` looks up the given attribute in the element's attribute map. It returns `Some attr` if there exists an attribute with the given key, and `None` otherwise. The official specification also has a concept called "reflected content attribute," which basically returns the stored attribute of the same name, but returns the empty string if the attribute is not present.

The function `set_attribute` updates the given attribute of the pointer in the heap. In the official specification, it is not allowed to set the attribute to `None` or null, respectively, to delete the attribute. We generalize this definition by allowing this behaviour.

The function `get_parent_node` takes a pointer to a node and returns a pointer to its parent, or `None`, if the node does not have a parent. The case where a node does not have a parent can only occur in *disconnected* node trees, which we will discuss later. Our API does not accept `documents`, since they can never have a parent. Having the types as narrow as possible, i. e. allowing as few possible types as possible, will enable easier proofs. The function `get_parent_node` is an example of a method where the official specification leaves much room for interpretation regarding the implementation. It neither provides an algorithm explaining to how obtain a parent, given a node, nor does it specify that the parent reference should be stored in the objects. To avoid specifying additional consistency constraints that would be needed if both children and parent references were stored, we defined `get_parent_node` by searching the whole heap for any node whose

`get_child_nodes` contains the given reference.

The function `remove_child` is rather close to the official specification; if `child`'s `parent` is different from the passed pointer, we then "throw" a `NotFoundError`. Otherwise, we add the removed child to the disconnected node list of its owner document and remove it from either the `document_element` or the `child_nodes` attribute.

The function `get_element_by_id` searches in *tree order* (depth-first, left-to-right) for the first element with the given id. Our definition is more general than the official specification, as we dropped the requirement that `get_element_by_id` should only be available on documents, which is a legacy requirement.

The method `adopt_node` removes a node from its parent, if it had any, and assigns it to the new `ownerDocument` as a disconnected node. First, it tries to retrieve the parent of the node to be adopted. If the node has a parent node, it removes the node from the children list, otherwise it removes it from the list of disconnected nodes of the previous owner document. Finally, the node is now added to the disconnected nodes of the new document. Since this is a rather complicated, but also important method, we now show our formalisation of it:

```
definition adopt_node :: "(_) document_ptr ⇒ (_) node_ptr
    ⇒ (_, unit) dom_prog"
where
  "adopt_node document node = do {
    old_document ← get_owner_document (cast node);
    parent_opt ← get_parent node;
    (case parent_opt of
      Some parent ⇒ do {
        remove_child parent node
      } | None ⇒ do {
        return ()
      });
    (if document ≠ old_document then do {
      old_disc_nodes ← get_disconnected_nodes old_document;
      set_disconnected_nodes old_document
        (remove1 node old_disc_nodes);
      disc_nodes ← get_disconnected_nodes document;
      set_disconnected_nodes document (node # disc_nodes)
    } else do {
      return ()
    })
  }"
```

Summary

So far, we have defined the core of what makes up *fDOM*; the underlying data structures to represent node trees and the functions to create, inspect, and modify them. Most of

the design decisions have been made to try and stay as close to the DOM standard as possible. However, the heap datatype is not restrictive enough yet to accurately capture the constraints of a node tree. These kinds of restrictions can not be easily pushed into the used types, however, so we will look into what kinds of invariants are still missing in the next section.

3.3 Well-Formedness of the DOM Heap

Our DOM heap is a finite map from pointers to objects. A finite map alone, however, would allow numerous "illegal" heaps. Two features of our formalisation rule out many, but not all, misconfigurations: Our data model is typed and, thus, rules out illegal heaps such as one that contains a document with a character data object as its only child. Additionally, our data model omits some fields of the standard, such as `parentNode`, which we calculate by using the heap and `get_child_nodes`.

Still, some possible illegal heap configurations remain. For example, a heap could still have a cyclic `get_child_nodes` relationship, which would result in a graph that is not a tree anymore. Thus, we need further well-formedness constraints. Additionally, we need to show that the DOM methods preserve these well-formedness constraints, which is why we will also call these constraints *invariants*. We will now introduce predicates that validate whether a given heap conforms to the standard.

This is not just necessary for our formalisation; by doing so we uncover invariants that also need to be obeyed by the official standard, even though they are not mentioned there, not even within non-normative sections. The invariants are only being preserved *implicitly* by the DOM methods.

3.3.1 Node Sharing

The DOM standard assumes that a node cannot be the child of more than one node. This property of heaps is informally implied by the official standard, and all tree-modifying methods ensure that such a DOM cannot be built. We, however, must deal with all heaps that conform to our heap type. Therefore, we formally define the following heap predicate:

```
definition distinct_lists :: "(_) heap ⇒ bool"
where
  "distinct_lists h = distinct (concat (
    (map (λptr. |h ⊢ get_child_nodes ptr|ᵣ)
      |h ⊢ object_ptr_kinds_M|ᵣ)
  @ (map (λdocument_ptr. |h ⊢ get_disconnected_nodes document_ptr|ᵣ)
      |h ⊢ document_ptr_kinds_M|ᵣ)
  ))"
```

The definition first builds a list of all children and disconnected nodes in any objects in the current heap and then asserts that this list does not contain any duplicates by using the `distinct` predicate. This not only prevents parents from sharing children or

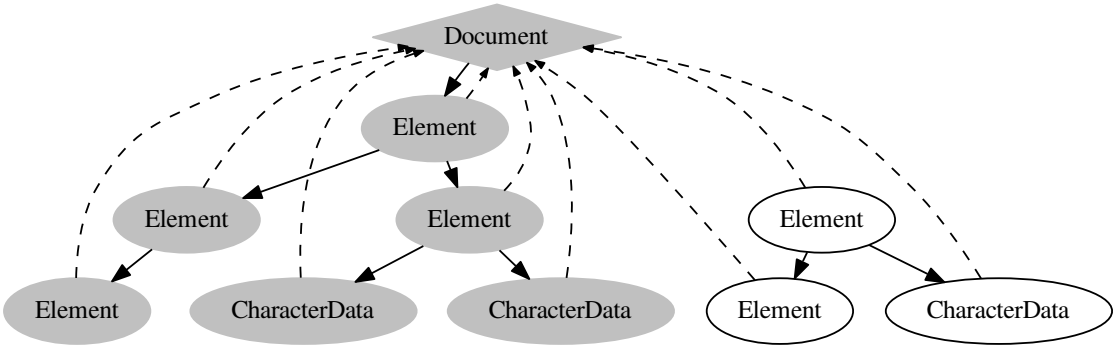


Figure 3.9: A schematic DOM instance with a visible document (gray) and a disconnected runtime tree (white).

documents from sharing disconnected nodes, but also ensures that no nodes are child and disconnected node at the same time. In addition, this predicate ensures that we do not have duplicate pointers in our lists, which is important for the proper functioning of DOM methods such as removing a child.

3.3.2 The Owner Document

The DOM specifications requires that each node is owned by exactly one document: its owner document. Moreover, each node participates in a tree with regards to the `get_child_nodes`-relation. A DOM might–and usually will–consist of several trees, i. e., a DOM is a forest of trees. We call the tree that has the main document as root the *visible document*, as this is the part of the DOM that would be rendered, e. g., by a web browser.

Figure 3.9 illustrates this relationship for an example: the gray nodes (connected to the node document by solid arrows that visualize the `get_child_nodes`-relation) represent the visible document. The white nodes (connected to the node document by dotted arrows that visualize the `get_owner_document` relationship) are forming a temporary *runtime tree*. Runtime trees are not serialized (e. g., in an HTML or XML document) and only exist at runtime.

We define `get_owner_document` of a node to be the root, if the root is a document; otherwise, we return that document whose `disconnected_nodes` contains the given node. In order for this definition to be well-formed, we need the following predicate:

```
definition owner_document_valid :: "(_) heap ⇒ bool"
where
  "owner_document_valid h ⟷ (∀node_ptr ∈ fset (node_ptr_kinds h).
    ((∃document_ptr. document_ptr |∈| document_ptr_kinds h
       ∧ node_ptr ∈ set |h ⊢ get_disconnected_nodes document_ptr|ᵣ)
  ∨ (∃parent_ptr. parent_ptr |∈| object_ptr_kinds h
         ∧ node_ptr ∈ set |h ⊢ get_child_nodes parent_ptr|ᵣ)))"
```

This predicate, together with `distinct_lists`, guarantees that the set of nodes in all `disconnected_nodes` fields is exactly the set of nodes that do not have a document as their

root. It asserts that for each `node_ptr` in the heap, we either have a `document_ptr` that is in the heap and has `node_ptr` as a disconnected node, or that we have a `parent_ptr` that is in the heap and has `node_ptr` as one of its children. The `distinct_lists` predicate ensures this *or* is actually an *exclusive or*.

3.3.3 Restricting DOMs to Trees

So far, we do not restrict the relation given by `get_child_nodes` to be *acyclic*, which is possible since we use pointers. To prevent this, we can use the following definitions:

```
definition parent_child_rel :: "(_) heap
    ⇒ ((_) object_ptr × (_) object_ptr) set"
where
  "parent_child_rel h = {(parent, child). parent |∈| object_ptr_kinds h
    ∧ child ∈ cast ` set |h ⊢ get_child_nodes parent|_r}"

definition acyclic_heap :: "(_) heap ⇒ bool"
where
  "acyclic_heap h = acyclic (parent_child_rel h)"
```

First, we define `parent_child_rel`, which returns a relation of all parent-child pairs in the heap. In other words, it is the set of all solid black arrows in Figure 3.9. Then, we leverage the definition of acyclicity on relations, `acyclic`, to assert that all parent-child pair in a given heap `h` indeed represent an acyclic relation.

3.3.4 Pointer Validity

Moreover, we need to ensure that objects do not contain invalid pointers, i. e., pointers that do not point to an object stored in the heap. Otherwise, whenever we work with our pointers, we would have to deal with the possibility of a "null-pointer exception". Thus, we require:

```
definition all_ptrs_in_heap :: "(_) heap ⇒ bool"
where
  "all_ptrs_in_heap h ⟷
    (∀ptr ∈ fset (object_ptr_kinds h).
      set |h ⊢ get_child_nodes ptr|_r ⊆ fset (node_ptr_kinds h)) ∧
    (∀document_ptr ∈ fset (document_ptr_kinds h).
      set |h ⊢ get_disconnected_nodes document_ptr|_r
        ⊆ fset (node_ptr_kinds h))"
```

The only place where we can find pointers (without arbitrarily constructing them, which should be avoided) is in one of the datatype fields. Therefore, for all pointers in the heap, we retrieve the corresponding object, and check whether all pointers stored in applicable fields (`childNodes`, `document_element`, and `disconnected_nodes`) are present in the heap.

3.3.5 Heaps are Strongly Typed

As we model typed pointers and objects, we want to assure that a pointer of a certain type actually maps to an object of the related type in a given heap, e. g., that a `document_ptr` actually maps to a document. The following predicate assures that this holds for the whole heap:

```
definition type_wf :: "(_) heap ⇒ bool" where
  "type_wf h = (CharacterDataClass.type_wf h ∧
  (∀document_ptr ∈ fset (document_ptr_kinds h).
    get_Document document_ptr h ≠ None))"
```

This predicate is built alongside our type universe, which means that when we are building the `Document` extension, we only need to check for all document pointers, in addition to the predicate of the previous universe extension, which is `CharacterDataClass` in this example. For more information about this extension mechanism, see Section 3.1.1. We check that for all pointers of a given type in the heap, we actually are able to retrieve an object of the appropriate type.

3.3.6 Well-Formed Heaps

To put it all together, we define a well-formed heap as a heap that satisfies all discussed constraints:

```
definition heap_is_wellformed :: "(_) heap ⇒ bool" where
  "heap_is_wellformed h ⟷
    acyclic_heap h ∧ all_ptrs_in_heap h
    ∧ distinct_lists h ∧ owner_document_valid h"
```

This combined predicate is of central importance to *fDOM*. It encapsulates the most important properties of a well-formed node tree. These properties are not explicitly described in the official standard, but only implied by the specifications of the methods, which contain some checks to ensure that, for example, no cycles are introduced into the tree. This formal predicate now gives us one possible set of such explicit invariants. They are immensely useful for many purposes, such as generating useful test cases that can check in isolation whether a given implementation respects all invariants.

However, we still need to give a formal justification for why this set of predicates deserve to be called invariants. In the next section, we will see that these predicates provide us assumptions for our functional correctness proofs that are strong enough, as well as that all of our DOM methods do actually preserve them.

3.4 Reasoning over the DOM

Up to this point, we have mainly set up the infrastructure that we need to reason about the DOM in Isabelle: we defined new syntax, types, and program definitions. For this, we could have used any normal functional programming language. In the following, we

will use the Isabelle/HOL prover to conduct many different kinds of proofs over the basis that we have, which is one of the benefits of using such a tool: combining high-level and easy-to-read functional-style definitions with proofs of functional correctness in a powerful higher-order logic. We will classify the DOM methods according to whether they are side-effect free and which heap locations (identified by pointer and attribute of the object) they read, or which heap locations they modify if they do so. Together with more specialized lemmas about the DOM methods, we will then show that all DOM API methods do indeed preserve all our invariants and how they do so. In addition, we will see that a number of lower-level methods sometimes break these invariants, highlighting the need to carefully consider which API to expose in order to ensure that the consistency and well-formedness of the heap is preserved.

It is important to recall that our focus here lies on the DOM and important properties that we are able to prove about DOM methods. While we touch on some topics that have been discussed in the area of separation logic for many years, such as "locality" of DOM methods or specifying exactly what parts of the heap are read and written by them, ultimately we do not need their full power, complexity, and level of automation to reach our goal: creating a lightweight proving-infrastructure of DOM methods and web components. In the future, however, it might well be worth the effort to extend, e. g., a powerful Abstract Separation Logic framework such as Smallfoot [65] with support for richer heaps and methods to underpin our *fDOM* in order to gain a higher degree of proof automation.

3.4.1 Locality of Heap Modifications

Before we can start to reason about the behavior of whole programs, we need to have a closer look at how our heaps change during the execution of our programs. Most heap-modifying functions do not modify the heap *arbitrarily*. Many functions even set only one attribute of only one object, i. e., setters such as `set_attribute`. Others are more complicated; methods such as `insert_before` modify attributes of multiple objects, adding and deleting nodes from lists. Still, the modified attributes and objects can be precisely specified.

When it comes to pure, or "read-only", functions, we might not need to worry about how the heap is modified, but characterizing the result can also be tricky. For example, after calling `insert_before`, the new child node surely will appear in the list of nodes returned by `get_child_nodes` on the same pointer. But what happens if we call `set_attribute` in between these two calls? Or what happens if we add another call of `insert_before`, but on a separate part of the heap?

To answer these questions, we first introduce two predicates that characterize a function by specifying which locations (pointers) and fields are being read or written, respectively:

```
definition reads :: "('heap ⇒ 'heap ⇒ bool) set ⇒ ('heap, 'e, 'result) prog
  ⇒ 'heap ⇒ 'heap ⇒ bool"
where
  "reads S getter h h' ⟷ (∀P ∈ S. reflp P ∧ transp P)
    ∧ ((∀P ∈ S. P h h') ⟶ preserved getter h h')"
```

```
definition writes :: "('heap, 'e, 'result) prog set ⇒ ('heap, 'e, 'result2) prog
  ⇒ 'heap ⇒ 'heap ⇒ bool"
where
  "writes S setter h h' ⟷ (h ⊢ setter →ₕ h'
    ⟶ (∃progs. set progs ⊆ S ∧ h ⊢ iterate_M progs →ₕ h'))"
```

The `reads` predicate takes a set of predicates and a getter function; the set of predicates basically encodes the conditions under which the given getter will behave in the same manner across different heaps. For example, `get_child_nodes` will return the same result if the `childNodes` and `documentElement` fields across all objects in heaps `h` and `h'` are the same. We can be even more specific: the method called on a pointer `ptr` will behave the same across two heaps if the fields of the object referenced by `ptr` do not change.

In contrast, the `writes` predicate works differently: it takes a set of programs and a setter function. The predicate then holds if the setter can be substituted for a chain of programs (executed by using `iterate_M`) from the given set `progs`. This then allows us to characterize a heap-modifying method by the pointers and attributes that are changed by it. For example, for the `get_attribute` and `set_attribute` DOM methods we prove the following two lemmas:

```
lemma get_attribute_reads:
  "reads {preserved (get_M element_ptr attrs)}
    (get_attribute element_ptr k) h h'"
```

This lemma says that a call of `get_attribute` only "reads" the `attrs` attribute of the element class, i. e., whenever we can show that the `attrs` attribute remains unchanged between two heaps `h` and `h'`, we know that `get_attribute` remains unchanged, too, and will exhibit the same behaviour on both heaps. Such lemmas will be very useful whenever we want to show that certain modifications do not affect certain getters.

```
lemma set_attribute_writes:
  "writes (all_args (put_M element_ptr attrs_update))
    (set_attribute element_ptr k v) h h'"
```

This lemma says that `set_attribute` only changes the `attrs` attribute of the element referred to by `element_ptr`, and makes no other changes to `h'` compared with `h`. We will use such lemmas mainly whenever we want to replace a complex DOM method by just the locations and attributes that it modifies.

We prove such lemmas for every one of our DOM methods. In essence, these lemmas characterize the behaviour of all methods by means of their use of getter (`get_M`) and setter (`put_M`) functions. In this example, `attrs` is the method provided by the record package to retrieve the `attrs` field from a record; `attrs_update` is used to set it.

Table 3.1 shows which methods access which attributes. We distinguish two kinds of accesses: fields of objects that are referenced by the function call, and fields of objects that are "further away", highlighting potentially unexpected field accesses. It also provides a measure for the complexity of the given method.

Table 3.1: DOM functions (top part) and internal helper functions (bottom part), and which attributes they read (if the function is pure) or write (if the function modifies the heap). A bold tick indicates that the attribute is not just read from the pointers explicitly passed to the function, but also from other pointers in the heap.

	RElement.tag_name	RElement.child_nodes	RElement.attributes	RCharacterData.data	RDocument.doctype	RDocument.document_element	RDocument.disconnected_nodes
get_child_nodes		✓				✓	
get_parent_node		**✓**				**✓**	
get_root_node		**✓**				**✓**	
get_element_by_id		**✓**	**✓**			**✓**	
get_owner_document		**✓**				**✓**	**✓**
get_attribute			✓				
get_data				✓			
get_tag_name	✓						
create_element	✓	✓	✓				✓
create_character_data				✓			✓
create_document					✓		✓
set_attribute			✓				
remove_child		✓				✓	**✓**
adopt_node		**✓**				**✓**	✓
insert_before		**✓**				**✓**	**✓**
append_child		**✓**				**✓**	**✓**
set_child_nodes		✓					
get_disconnected_nodes							✓
set_disconnected_nodes							✓
set_tag_name	✓						
set_data				✓			

3.4.2 Exceptions and Non-Termination

All our functions can throw exceptions, i. e., they return a sum type of `exception` and their real return type, which is a common way of modelling exceptions in functional languages. Therefore, we can provide lemmas that show under which preconditions our functions will return their normal result and not throw an exception. Most DOM functions will throw an exception under exactly one circumstance: if they try to resolve a pointer on the given heap which does not have an object of the same type stored in that location, i. e., the lookup returns `None`. This is expected, since most functions will need to do something with the object, and not just the pointer to the object. For example, we show:

```
lemma set_attribute_ok:
  assumes "type_wf h"
  assumes "element_ptr |∈| element_ptr_kinds h"
shows "h ⊢ ok (set_attribute element_ptr k v)"
```

The method `set_attribute` does not return an error if the heap is well-typed, i. e., the types of all pointers and their objects match, and if the argument `element_ptr` to the function is present in the heap.

Another, less common source of errors is non-termination. As we have a heap structure with pointers, any function that iterates along the graph could potentially loop infinitely. Normally, Isabelle provides enough automatic support that termination need not be worried about for simple recursive methods, e. g., methods that simply "unpack" datatypes. In our case, however, it is not always obvious why a function should terminate. In heaps that do not fulfil our well-formedness predicates, there are potentially many situations where a function that iterates along the node graph does not terminate. Our heap datatype alone is not sufficient to prevent these cases, as we have seen earlier.

We define recursive DOM methods by using Isabelle's `partial_function` command. Despite the name, the resulting function is still total, but maps the case of non-termination to a special datatype instead of preventing the definition altogether. Naturally, Isabelle is unable here to generate many of the useful equations for us that we get for, e. g., `definition`, since many of them now require additional preconditions. For example, the formalisation of the "tree order" of the standard looks as follows:

```
partial_function (dom_prog) to_tree_order :: "(_) object_ptr
  ⇒ (_, (_) object_ptr list) dom_prog"
where
  "to_tree_order ptr = do {
    children ← get_child_nodes ptr;
    treeorders ← map_M to_tree_order (map cast children);
    return (ptr # concat treeorders)
  }"
```

Isabelle allows us to define many kinds of recursive functions that way without the need to worry about termination (during the definition). We configured `partial_function` in a way that effectively maps the case of non-termination to a specific value of our

`dom_prog` type–more specifically, the `NonTerminationException`. Of course, to prevent inconsistencies in the logic, we only get few lemmas to work with from this kind of definition. Therefore, before we can use this kind of definition in proofs, we need to show that the given method call does not return an error, for which we use the `ok` predicate from our monad theory. These proofs are more complicated now, though, and involve the use of the heap well-formedness predicates (guaranteeing an ayclic node tree) in conjunction with well-founded induction. In more detail, one of the key lemmas for our termination proofs is the following:

```
lemma heap_wellformed_induct:
  assumes "heap_is_wellformed h"
  assumes step:
    "⋀parent. (⋀children child. h ⊢ get_child_nodes parent →ᵣ children
      ⟹ child ∈ set children ⟹ P (cast child)) ⟹ P parent"
  shows "P ptr"
```

It basically says that whenever we have a wellformed heap `h` we can proof any property `P` by induction over the children relationship, i. e. if we can show that when a property `P` holds for all children of a pointer `parent` it also holds for `parent` itself, then `P` holds for *any* pointer. This induction lemma itself itself holds because the heap-wellformedness provides us with a well-founded relation `parent_child_rel` (which is built from `get_child_nodes`), which we use to leverage transfinite induction on the children relationship.

3.4.3 Functional Correctness

One goal of our formalisation of the DOM is to prove the functional correctness of the most common API methods. Especially complex functions such as `adopt_node` and `insert_before`, which modify the heap in a non-local way, are of interest, because they have the rather surprising behaviour of removing a node first before inserting another node. This behaviour also has the potential to leave a misshapen node tree. Doing this analysis in Isabelle/HOL also has the advantage that the look of the properties are relatively close to how they could be presented, for example, as part of the WebIDL specifications in the official standard.

In the following, we will show two of many important properties about `insert_before`. First, we will show that using `insert_before` never leads to duplicates among the node's children, even if a pointer is being inserted that is already in this node's children. Second, we will show that `insert_before` also never introduces cycles into the node tree. Both properties are important for the well-formedness invariants in Section 3.3.

We start by showing that `insert_before` never leads to duplicates in a child node list. Since we use HOL lists (as opposed to, e. g., ordered sets), which can contain duplicate pointers, the type alone does not provide the desired constraint. Therefore, to show that if we start with a well-formed heap, we will never end up with duplicate pointers, we show the following lemma:

```
lemma insert_before_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
```

```
  assumes "h ⊢ insert_before ptr new_child child_opt →h h'"
shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →r children' ⟹ distinct children'"
```

The conclusion is to be read as follows: After the use of `insert_before` (represented by state `h'`), all lists of children of all pointers will be distinct. To prove such a property about the behavior of the DOM method, we first recall its definition earlier. It contains two crucial heap-modifying methods: `adopt_node` and `insert_node`, an auxiliary method that is not part of the standard, but which we use to facilitate the actual insertion of the node into the list. Now, fortunately the lemma that we intend to prove also holds for these two methods.

For `adopt_node`, the situation is relatively simple, since the method only removes a child:

```
lemma adopt_node_removes_child:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ adopt_node owner_document node_ptr →h h'"
shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →r children' ⟹ node_ptr ∉ set children'"
```

Therefore, we can immediately continue to show that `adopt_node` never introduces duplicates:

```
lemma adopt_node_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ adopt_node owner_document node_ptr →h h'"
shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →r children' ⟹ distinct children'"
```

Since `adopt_node` only ever removes nodes from lists of children, this property is simpler to prove. Showing that property for `insert_node`, however, is more challenging:

```
lemma insert_node_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_node ptr new_child reference_child_opt →h h'"
  assumes "h ⊢ get_child_nodes ptr →r children"
  assumes "new_child ∉ set children"
shows "⋀children'.
  h' ⊢ get_child_nodes ptr →r children' ⟹ distinct children'"
```

Since `insert_node` inserts the node without any checks, we need to introduce the additional assumption that `new_child` was not already in the list of children beforehand. Equipped with these three lemmas, we can go back to prove our original lemma. This is a common proof schema that we use for many DOM methods; if we want to prove a property for a method, we can then prove said property for each other method that the definition depends on. Once that is done, we can then prove the original lemma by first fixing all necessary variables, usually using **`obtain`**, and then establishing the property step by step starting from the original state `h` to the final state `h'`. To keep already

established properties across these different heaps, the `reads` and `writes` lemmas from Section 3.4.1 are especially useful. Using this technique, we now prove that the children of all nodes remain distinct after `insert_before` as been called anywhere in the heap:

```
lemma insert_before_children_remain_distinct:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr new_child child_opt →ₕ h'"
shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →ᵣ children' ⟹ distinct children'"
proof −
  obtain reference_child owner_document h2 h3 disconnected_nodes_h2 where
    reference_child: "h ⊢ (if Some new_child = child_opt then
       next_sibling new_child else return child_opt) →ᵣ reference_child" and
    owner_document: "h ⊢ get_owner_document ptr →ᵣ owner_document" and
    h2: "h ⊢ adopt_node owner_document new_child →ₕ h2" and
    disconnected_nodes_h2:
      "h2 ⊢ get_disconnected_nodes owner_document →ᵣ disconnected_nodes_h2" and
    h3: "h2 ⊢ set_disconnected_nodes owner_document
      (remove1 new_child disconnected_nodes_h2) →ₕ h3" and
    h': "h3 ⊢ insert_node ptr new_child reference_child →ₕ h'"
    using assms(4)
    by ...
  have "⋀ptr children. h2 ⊢ get_child_nodes ptr →ᵣ children
      ⟹ distinct children"
    using adopt_node_children_remain_distinct ...
    by blast
  moreover have "⋀ptr children. h2 ⊢ get_child_nodes ptr →ᵣ children
      ⟹ new_child ∉ set children"
    using adopt_node_removes_child ...
    by blast
  moreover have "⋀ptr children. h2 ⊢ get_child_nodes ptr →ᵣ children
      = h3 ⊢ get_child_nodes ptr →ᵣ children"
    using get_child_nodes_reads set_disconnected_nodes_writes h3
    by ...
  ultimately show "⋀ptr children. h' ⊢ get_child_nodes ptr →ᵣ children
      ⟹ distinct children"
    using insert_node_children_remain_distinct
    by ...
qed
```

First, we unfold the definition of `insert_before` to get a handle to the individual statements. Additionally, we obtain `h2` and `h3`, which are the intermediate heap in between `adopt_node` and `set_disconnected_nodes` and in between `set_disconnected_nodes` and `insert_node`, respectively.

Second, we prove that after `adopt_node`, in addition to all children lists still being

distinct (we only remove one child from one list), the child will not be part of any of these as it has been removed from the only children list that contained it. For this proof, we can use two properties of `adopt_node` that we proved earlier, `adopt_node_removes_child` and `adopt_node_children_remain_distinct`.

Third, since we know that before the use of `insert_node` all children lists are distinct and do not contain the node to be inserted, we can prove that all children lists will remain distinct, as we only insert the given node and nothing else. Again, for this proof, we can use a previously proven property about `insert_node`; `insert_node_children_remain_distinct`.

Now, we move on to a property about `insert_before` that is more complicated: We want to analyse exactly how the specification of `insert_before` ensures that it does not introduce any cycles into the node tree as an example of a functional correctness proof. For this purpose, we will look at various lemmas to help us prove two important observations: After calling `insert_before`, we will 1. never have a node twice in the node tree, and 2. receive an error if we try to insert a node into the children of one of its descendants.

The first observation is achieved by `insert_before` removing nodes from its previous parent before inserting it into the list of children:

```
lemma insert_before_removes_child:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr node child →ₕ h'"
  assumes "ptr ≠ ptr'"
shows "⋀children'.
  h' ⊢ get_child_nodes ptr' →ᵣ children' ⟹ node ∉ set children'"
```

We again fix two heaps, `h` and `h'`, representing the state before and after the call to `insert_before`, respectively. Then, we fix a pointer `ptr'` that is different from `ptr` on which we called the method. Finally, we show that in state `h'`, this other pointer `ptr'` certainly does not have `node` as one of its children. Note that we do not make any further assumptions, besides our usual well-formedness predicates. It does not matter whether `node` was previously child of some other node, or perhaps a disconnected node. The specification of `insert_before` ensures that in any case, the node does not appear anywhere in the node tree besides as part of the children of `ptr`, thus preventing any duplicate occurrence of `node`.

The removal of the node prior to insertion is actually achieved by calling `adopt_node` in the specification of `insert_before`, which removes a given node from the node tree and places it in the list of disconnected nodes, which we have already seen for the previous example.

Before we show the second observation, we will have a closer look at the "pre-insertion validity" from the standard, which we give verbatim in Figure 3.7 and as an Isabelle function in Figure 3.10. For us of interest is especially Line 7 until Line 9, which retrieves the list of ancestors of the reference node `parent`, and then returns a `HierarchyRequestError` if the node `node`, which is to be inserted, is already an ancestor of `parent`. The intention of this check is to prevent the introduction of cycles, although it is never clearly specified, and it is not clear whether that is enough, on its own, to achieve that goal.

```
definition ensure_pre_insertion_validity :: "(_) node_ptr
  ⇒ (_) object_ptr ⇒ (_) node_ptr option ⇒ (_, unit) dom_prog"
where
  "ensure_pre_insertion_validity node parent child_opt = do {
    (if is_character_data_ptr_kind parent
      then error HierarchyRequestError else return ());
    ancestors ← get_ancestors parent;
    (if cast node ∈ set ancestors
      then error HierarchyRequestError else return ());
    (case child_opt of
      Some child ⇒ do {
        child_parent ← get_parent child;
        (if child_parent ≠ Some parent
          then error NotFoundError else return ())}
    | None ⇒ return ());
    children ← get_child_nodes parent;
    (if children ≠ [] ∧ is_document_ptr parent
      then error HierarchyRequestError else return ());
    (if is_character_data_ptr node ∧ is_document_ptr parent
      then error HierarchyRequestError else return ())
    }"
```

Figure 3.10: The formalisation of "pre-insertion" validity (Figure 3.7) in Isabelle/HOL as a program that terminates correctly if fulfilled, and throws the appropriate exception if not.

Finally, with these two observations proven, we can attempt to prove the lemma that we maintain an acyclic heap, i. e., the relation `parent_child_rel h` remains acyclic:

```
lemma insert_before_preserves_acyclitity:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr node child →ₕ h'"
shows "acyclic (parent_child_rel h')"
```

To prove this property, our approach of simply lifting the proofs from the "smaller" definitions to the more complicated ones does not work, as the unchecked node insertion of `insert_node` certainly *could* lead to cycles. Therefore, we will now look at the proof skeleton, which uses lemmas related to relations (the full proof is, as always, part of the formalisation document):

```
1  proof −
2    obtain ancestors reference_child owner_document h2 h3
3      disconnected_nodes_h2
4    where
5      ancestors: "h ⊢ get_ancestors ptr →ᵣ ancestors" and
6      node_not_in_ancestors: "cast node ∉ set ancestors" and
7      reference_child:
8         "h ⊢ (if Some node = child then next_sibling node
9           else return child) →ᵣ reference_child" and
10     owner_document: "h ⊢ get_owner_document ptr →ᵣ owner_document" and
11     h2: "h ⊢ adopt_node owner_document node →ₕ h2" and
12     disconnected_nodes_h2: "h2 ⊢ get_disconnected_nodes owner_document
13       →ᵣ disconnected_nodes_h2" and
14     h3: "h2 ⊢ set_disconnected_nodes owner_document
15       (remove1 node disconnected_nodes_h2) →ₕ h3" and
16     h': "h3 ⊢ insert_node ptr node reference_child →ₕ h'"
17     by ...
18   ...
19   have "cast node ∉ {x. (x, ptr) ∈ (parent_child_rel h2)*}"
20     using adopt_node_removes_child
21     using ancestors node_not_in_ancestors
22     using ...
23     by blast
24   then have "cast node ∉ {x. (x, ptr) ∈ (parent_child_rel h3)*}"
25     by ...
26   moreover have "parent_child_rel h'
27       = insert (ptr, cast node) (parent_child_rel h3)"
28     by ...
29   ultimately show "acyclic (parent_child_rel h')"
30     by ...
31 qed
```

The proof script has two main parts: In the upper half, we use the **obtain** command to divide the definition of `insert_before` into its parts so we can reference them in the proof. Most importantly, we introduce the intermediate heaps `h2` and `h3` so we can reference the changes to the heap in a more fine-grained manner. We reference our first observation by lemma name in Line 20 and Line 21, whereas the second observation follows directly from the definition of `insert_before`, so it is part of the **obtain** construction in Line 5 and Line 6. Then, the proof script contains two critical steps: in Line 19, we prove that after calling `adopt_node` (in `h2`), `node` does not appear as a parent anywhere in the heap. Afterwards, Line 26 characterizes our helper method `insert_node`, which does the actual work of inserting the node into the list. After it, we have a new pair `(ptr, cast node)` as part of our `parent_child_rel` relation. As we know that it did not contain any other pair ending with `cast node`, we can finally conclude that we have not introduced a cycle.

3.4.4 Well-Formedness of the Heap Methods

With the groundwork done, we can now tackle the goal of verifying that all DOM API methods preserve the heap well-formedness invariants. This would also mean that any exception-free sequence of DOM methods creates a well-formed DOM heap.

To achieve this goal, we need to prove a lemma of this form for all methods:

```
lemma insert_before_heap_is_wellformed_preserved:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "h ⊢ insert_before ptr node child →ₕ h'"
  shows "heap_is_wellformed h'" and "known_ptrs h'" and "type_wf h'"
```

All variables in lemmas are all-quantified, meaning they can take all possible values of their types, only restricted by the **assumes** statements. As the predicate `heap_is_wellformed` is a conjunction of more specific predicates (e. g., `acyclic_heap`), we can prove that they are preserved separately. We have shown two of such proofs in the previous section. In addition, Table 3.2 shows a more detailed view, listing which DOM methods preserve which invariant. While the DOM API methods all do so, not all of our "internal" functions do, demonstrating why many more primitive methods are missing from the DOM API.

Summary

In this chapter, we have built a formal model of the core DOM, the *fDOM*, in Isabelle/HOL. We have seen a formal description of the properties of the *node-tree*, the central data structure at the heart of the DOM. One of the shortcomings of the official standard is the lack of properties that hold for the DOM methods, making the semi-formal descriptions difficult to grasp. The well-formedness predicates that we have seen in this section improve this situation by providing a concise specification of these implicit assumption, all founded in higher-order logic. Additionally, the *fDOM* provides us with an analysis framework that we will use in the following chapters to analyse further properties in this style of *shadow roots*, as well as concepts that lie beyond what is specified in the standard and will result in our proposal for safer web components.

Table 3.2: DOM functions (top part) and internal helper functions (bottom part), and which well-formedness predicates they preserve. We added functions that we use internally in *fDOM*, but are *not* part of the DOM API, to demonstrate the problem that would arise if all getter and setter functions part of the DOM API.

	acyclic_heap	all_ptrs_in_heap	distinct_lists	owner_document_valid	type_wf	known_ptrs
get_child_nodes	✓	✓	✓	✓	✓	✓
get_parent_node	✓	✓	✓	✓	✓	✓
get_root_node	✓	✓	✓	✓	✓	✓
get_element_by_id	✓	✓	✓	✓	✓	✓
get_owner_document	✓	✓	✓	✓	✓	✓
get_attribute	✓	✓	✓	✓	✓	✓
get_data	✓	✓	✓	✓	✓	✓
get_tag_name	✓	✓	✓	✓	✓	✓
create_element	✓	✓	✓	✓	✓	✓
create_character_data	✓	✓	✓	✓	✓	✓
create_document	✓	✓	✓	✓	✓	✓
set_attribute	✓	✓	✓	✓	✓	✓
remove_child	✓	✓	✓	✓	✓	✓
adopt_node	✓	✓	✓	✓	✓	✓
insert_before	✓	✓	✓	✓	✓	✓
append_child	✓	✓	✓	✓	✓	✓
set_child_nodes						
get_disconnected_nodes	✓	✓	✓	✓	✓	✓
set_disconnected_nodes	✓					
set_tag_name	✓	✓	✓	✓	✓	✓
set_data	✓	✓	✓	✓	✓	✓

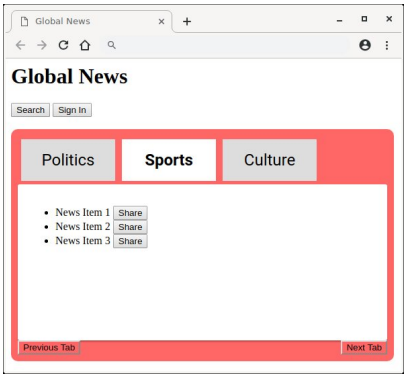
4 A DOM with Shadow Roots

Now that we have established a formal foundation of the DOM, we can focus on exploring those parts of the DOM specification that are suitable for component-based programming. While there are also parts that are relevant for *using* components in other specifications, e. g., the HTML standard, the fundamental mechanisms for that are (now) part of the DOM standard. To be more specific, what was previously called "Shadow DOM" and has now been integrated into the DOM, is basically the concept of *shadow roots*. A shadow root is a new type of root node, similar to a `Document`, but it has to be "embedded" into an `Element` and thus begins a new kind of sub-tree starting from this `Element`. In a way, this additional kind of node adds another "dimension" to the node tree that we thoroughly discussed in the previous chapter. If we imagine the node tree as a two-dimensional plane, shadow roots allow us to extend that tree into a third dimension by layering node trees into related, but separate sub-trees. However, the new data model is only the smaller part of the changes that we will see in this chapter. We will also introduce new DOM API into *fDOM* that will make use of these shadow roots, handling them differently than the rest of the tree and thus enabling a kind of behaviour that we set out to find: parts of the node tree that are shielded from being manipulated from certain DOM methods, enabling components at the fundamental level. All Isabelle definitions and proofs of this chapter can be found online [14].

In this chapter, therefore, we will present the following contributions towards our goal of a formally verified model of components at the DOM level (i. e., components that are concerned with nodes and their relationship to each other):

First, we introduce an extension to *fDOM* containing the data model and DOM methods that are necessary to represent shadow roots. We will do this as a clean extension, meaning that we will not need to make any changes to the proof documents of the previous chapter. Using some features of Isabelle/HOL such as *locales*, we will be able to maintain the previous definitions and proofs with minimal effort in our new type universe–an enormous boon to the readability and maintainability of such large proof documents.

Second, we present a formalisation of the new algorithms that come with the introduction of shadow roots, along with the verification of the functional correctness of them. Most notably, this includes a formalisation of the "rendering view" as far as shadow roots are concerned, which is similar to what a web browser might render, in order to facilitate a better analysis of related algorithms. The standard introduces new concepts such as *slotting* that are responsible for the new behaviour of the DOM API that make components possible. It is a relatively new and complex concept and, unfortunately, described in the standard only in an imperative way, lacking any sort of guarantees that can be obtained by using them. Gaining a better understanding of them is therefore



(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

Figure 4.1: Our running example: *Fancy Tab*, a reusable component for tab-based navigation layouts.

crucial to our goal, and a formalisation of them will also prove useful later when we formalise our definition of web components.

4.1 Motivating Example

A shadow root is a new kind of node, similar to a `Document`, but with two key differences: First, shadow roots get embedded inside an `Element` (its *host*), instead of being a free-standing root node like a `Document`. Second, shadow roots can have a special kind of child element, a *slot*, which receives certain elements from other sub-trees as *temporary* children–only for rendering purposes in a final web page, however, no roots get actually moved. By using shadow roots, a library producer can develop reusable and consumer-customizable web components using HTML and a few lines of JavaScript.[1] In order to demonstrate the various concepts involved in using shadow roots, we will now introduce a small running example: *Fancy Tab*. It is multi-tab navigation view based on a simplified version of a similar widget found in [6].

Figure 4.1 demonstrates Fancy Tab: Figure 4.1a shows the end result, what the end-user will see in their browser, while Figure 4.1b shows how the consumer (the developer using the widget in his application) embeds the Fancy Tab HTML widget. To make it clearer what parts of the UI internally belongs to the component, we color those parts in red. For example, the two buttons on the bottom, “previous Tab” and “next Tab”, belong to Fancy Tab, as those kind of buttons could be seen as functionality that is common to all kinds of tab-based navigation widgets. Our implementation of Fancy Tab provides a custom HTML tag `<fancy-tabs>`, which takes its children and rearranges them according to their “slot” attribute through a process called *slotting*. Shadow roots

[1] While shadow roots are specified within the DOM standard, which naturally contains no references to HTML or JavaScript, we are not aware of any implementation other than those in web browsers using HTML and JavaScript.

additionally provide separation of executed client-side code: code running in the main web page should not interfere with code running in the context of Fancy Tab and vice versa. For example, widget code that changes the style attributes of the "previous Tab" and "next Tab" buttons in the lower corners of the widget should not affect buttons belonging to any parts belonging to the library consumer. Similarly, code that changes the styles of buttons outside of Fancy Tab should not have any effect on its buttons, even in the case of duplicate identifiers.

Sadly, the DOM standard neither defines the concept of web components nor specifies the safety properties that web components should guarantee. Consequently, the standard does not discuss how and if the methods for modifying the node tree respect component boundaries either. Thus, shadow roots are only the very first step in defining a safe web component model. In particular, the DOM standard lacks a formal definition of web components and precise specifications of the safety guarantees they can provide.

Now, we will discuss the inner workings of our running example Fancy Tab from Figure 4.1 in more detail. Figure 4.2a focuses on the HTML part of defining Fancy Tab. As the DOM standard does not allow 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.

Figure 4.2b shows an attempt to provide the functionality of Fancy Tab *without* using shadow roots. While this alternative definition would show no visible difference to the end-user, it does not provide any form of run-time separation.

We now assume that a consumer of Fancy Tab, who is unfamiliar with its inner workings, would like to change the appearance of some of their buttons. In particular, we assume that they would like to change all their button texts to upper case and thus use code similar to the snippet shown in Figure 4.3c.

Now, let us observe the results: Figure 4.3b shows the version without shadow roots; here, all buttons, including the navigation buttons on the bottom, became upper case, as they are part of the same "scope". We consider this undesired behavior, because the developer inadvertently modified the internal representation of Fancy Tab. However, to deserve the label "component", we would like the Fancy Tab developer to be protected from these kinds of effects.

Figure 4.3a shows the rendering of the version with shadow roots, where we can see that 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 scope `document`).

In order to understand the difference better, we will look at the DOM representation of our example with shadow roots, shown in Figure 4.4, which contains one additional small detail: a disconnected node, on the right-hand side of the figure, which could be present because, for example, the developer consuming Fancy Tab is currently creating new elements using JavaScript. By calling `document.getElementsByTagName("button")`, we enumerate all buttons, starting from the root document, and thus traverse the tree along the solid arrows (`childNode` relation). 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 lines (`shadowRoot` or `disconnectedNode` relation).

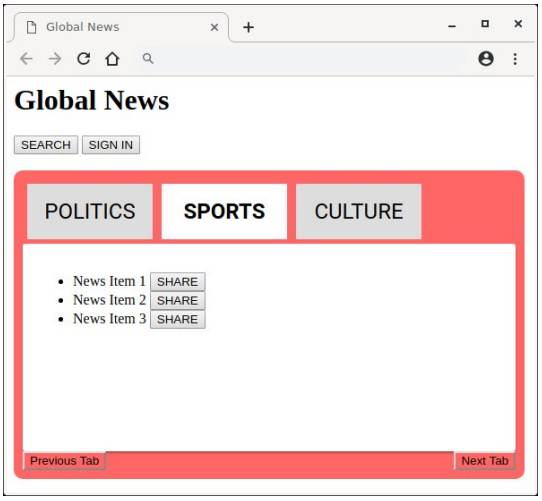
```
shadowRoot.innerHTML = '
  <style>...</style>
  <div id="tabs">
    <slot id="tabsSlot" name="title">
    </slot>









  </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.

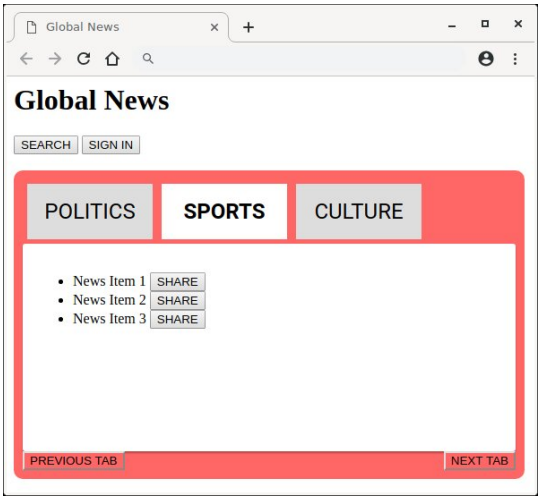
```
<div>
  <style>...</style>
  <div id="tabs">
    <button slot="title">
      Politics
    </button>
    <button slot="title" selected>
      Sports
    </button>
    <button slot="title">
      Culture
    </button>
  </div>
  <div id="panels">
    <section>
      content panel 1
    </section>
    <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 application, *losing any kind of separation properties.*

Figure 4.2: The source code of the Fancy Tab widget with shadow roots (left) and without (right).



(a) Styling Fancy Tab with shadow root only affects buttons outside of Fancy Tab.



(b) Styling fancy tabs without shadow root affects also 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.

Figure 4.3: Modifying a website that uses shadow roots versus one that does not.

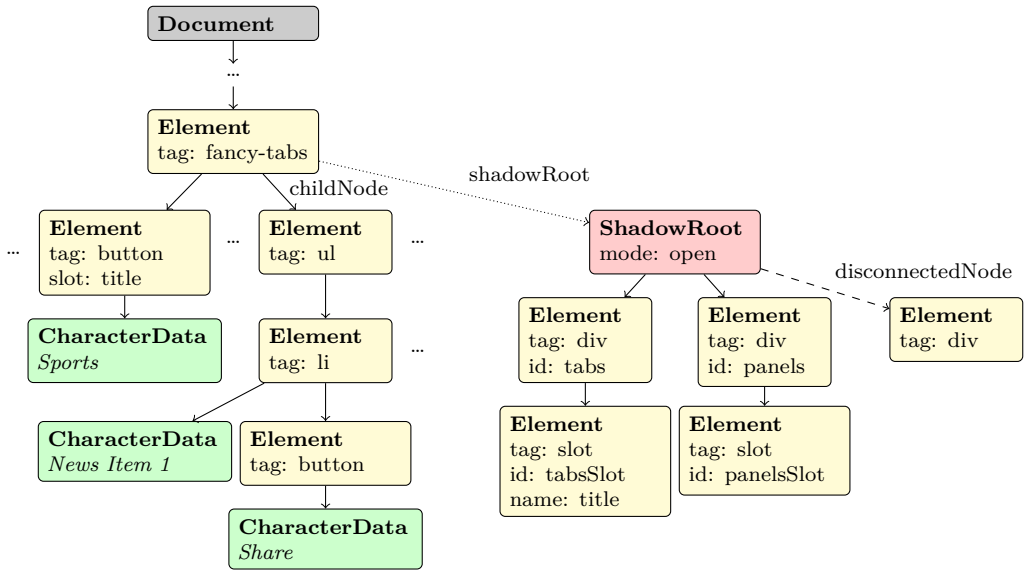


Figure 4.4: Representation of the internal DOM structure of our running example Figure 4.1, with the addition of a single disconnected node.

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

```
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"
```

Figure 4.5: The updated data model and DOM API specification including shadow roots.

To put it in a nutshell, the use of shadow roots to provide separation is orthogonal to the use of JavaScript. While there are plenty of patterns and mechanisms in the JavaScript ecosystem that provide some sort of separation, they can only separate *JavaScript methods and fields* from third-party code. When it comes to the access to the node tree, and the (possibly sensible) data that is stored in there, all JavaScript would have unrestricted access to read and modify. Therefore, shadow roots provide some kind of access control to already existing API methods by introducing new kinds of edges in the tree that are intentionally difficult to cross unless the developer already has access to pointers of this sub-tree.

4.2 Data Model and Basic Accessors

We will now use our *fDOM* from the previous chapter and extend it with support for shadow roots. First of all, we need to consider the data model, i. e., how shadow roots will be represented in a node tree. The official standard specifies that the shadow root is similar to a document in some ways, but is ultimately a different kind of node. Following the intuition from Figure 4.4, a shadow root represents the root node of a new kind of sub-tree, which is also called a *shadow tree*. A shadow root can also never exist on its own, as it must always be attached to an element. This element, in turn, is also special from the perspective of the shadow tree and is, together with its descendants, sometimes referred to as the *light tree*. These two terms are always relative to a given shadow root, so a shadow tree can indeed also contain a light tree of a different shadow root.

In order to extend the *fDOM* with such a new node type, we need to define a new record with new attributes. From a type hierarchy point of view, none of our existing types can be substituted by a shadow root, so we will define it at the top-most level right under objects. Using Web IDL, the interface of a shadow root is given in Figure 4.5.

Shadow roots can only be contained in an `Element`, where they behave as a special kind of child (successor) node. Instead of using the methods `get_child_nodes` and `get_parent`, we will introduce the methods `get_shadow_root` and `get_host`, respectively, that act similarly. The formal signatures of the basic methods for accessing shadow roots and their attributes are contained in Figure 4.5 as well.

A shadow root has a flag `mode` that indicates whether the sub-tree shall be accessible from the outside. For this purpose, the flag affects methods such as `get_shadow_root`, which will not return any node if the shadow tree is closed. If a shadow tree is closed, certain DOM methods can not access the shadow tree from the outside, i. e., `get_shadow_root` and `get_assigned_slot`. Figure 4.4 illustrates how the new node fits into the concept of a tree structure, as defined by the *fDOM*.

Additionally, a shadow root manages a list of children, exactly like an `Element` does, which is to be returned by the already existing `get_child_nodes` accessor. However, while the existing `get_child_nodes` method from *fDOM* takes indeed an `object_ptr`, if we were to pass a `shadow_root_ptr` to it, we would receive an error, since the existing implementation of `get_child_nodes` does not know yet about our newly introduced `childNodes` field of shadow roots. Therefore, we will make use of our extension mechanism as introduced previously, which means that we will extend the previous definition by creating a new one, with the same name, that calls the old one internally. Thanks to the extension mechanism, we will be able to keep code and proof duplication to a minimum.

In the following, we will explain this mechanism in more detail. Fundamentally, we use Isabelle's locales to create parametrised theories. Inside these locales, we define functions and prove lemmas as usual. Every function that we depend on, however, is not used directly, but is declared as a parameter to the locale. This is the *main idea* behind the way we use locales: when creating definitions and lemmas, we try to avoid referring to concrete definitions as much as possible. Instead, we will add and use locale assumptions that characterise the function only as much as necessary. As a result, because we have many DOM methods that might get extended later on, but keep their main behaviour intact, we can "generate" the appropriate lemmas for each such extension by, first, proving that the characterisations that we used remain valid, and, second, simply interpreting the locales with the new definition. Without that way of using locales we would have to re-state and re-prove many lemmas during the course of our extension, even though the function definitions barely change.

Additionally, we use a few other techniques as well that play into this pattern: For each DOM method, we create at least two locales: one containing the definitions, and another one containing the lemmas. This split helps to keep more control about introducing the definitions and lemmas into the global theory context when interpreting them later. Also, we create a helper function `invoke` that makes extending DOM functions easier by factoring out behaviour depending on the pointer type.

For example, when we extend the definition of `get_child_nodes`, we create a new locale $\texttt{l\_get\_child\_nodes}_{\texttt{Shadow\_DOM}}\texttt{\_defs}$, define the implementation of `get_child_node` for the case of shadow roots ($\texttt{get\_child\_nodes}_{\texttt{shadow\_root\_ptr}}$), and update the other methods relevant to our function definitions:

```
locale l_get_child_nodes_Shadow_DOM_defs =
  CD: l_get_child_nodes_Core_DOM_defs
begin
definition get_child_nodes_shadow_root_ptr :: "(_) shadow_root_ptr ⇒ unit
    ⇒ (_, (_) node_ptr list) dom_prog" where
```

```
  "get_child_nodes_shadow_root_ptr shadow_root_ptr _
    = get_M shadow_root_ptr RShadowRoot.child_nodes"

definition a_get_child_nodes_tups
  :: "(((_) object_ptr ⇒ bool) × ((_) object_ptr ⇒ unit
    ⇒ (_, (_) node_ptr list) dom_prog)) list" where
  "a_get_child_nodes_tups ≡ [(is_shadow_root_ptr_object_ptr,
    get_child_nodes_shadow_root_ptr ∘ the ∘ cast)]"

definition a_get_child_nodes :: "(_) object_ptr
    ⇒ (_, (_) node_ptr list) dom_prog" where
  "a_get_child_nodes ptr = invoke (CD.a_get_child_nodes_tups
    @ a_get_child_nodes_tups) ptr ()"

definition a_get_child_nodes_locs :: "(_) object_ptr
    ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set" where
  "a_get_child_nodes_locs ptr ≡
    (if is_shadow_root_ptr_kind ptr
    then {preserved (get_M (the (cast ptr)) RShadowRoot.child_nodes)}
    else {}) ∪
    CD.a_get_child_nodes_locs ptr"
end
```

We prefix many definitions with an `a_` which stands for *abstract*, because these definitions typically refer to other placeholder definitions instead of real ones. Only when we interpret the locale and thus fill in the "real" definitions, we will get a "non-abstract" definition in the Isabelle context. The `invoke` helper method acts as a kind of "virtual method table" and takes conditions and implementations in a list, allows us to simply append the implementation for shadow roots. This structure allows us to later split proofs into two cases: assuming that either the definition from the core DOM or the shadow root extension is called.

But first, we need to extend the main locale, $\texttt{l\_get\_child\_nodes}_{\texttt{Shadow\_DOM}}$. While the previous locale contained only definitions, this locale will contain lemmas. This split is done to improve the reuseability of lemmas, whereas with definitions that is not a concern. This locale depends on a number of other locales: on the one hand, the locales `l_type_wf` and `l_known_ptr` "import" the respective definitions–this time to be instantiated with the versions for the shadow root universe–, but on the other hand also on the main locale from the previous universe in order to leverage its already proven lemmas. Locales from the previous universe might also depend on locales such as `l_type_wf`, which we must not mix up with locales meant to be instantiated from the current universe (recall that we redefine some definitions each extensions), or otherwise we will not be able to instantiate our locale, as predicates such as `type_wf` or `known_ptr` are weaker in this universe. We therefore must rename all parameters stemming from core DOM locales. The locale specification for `get_child_nodes` with support for shadow roots looks as follows:

```
locale l_get_child_nodes_Shadow_DOM =
  l_type_wf type_wf +
  l_known_ptr known_ptr +
  l_get_child_nodes_Shadow_DOM_defs +
  l_get_child_nodes_defs get_child_nodes get_child_nodes_locs +
  CD: l_get_child_nodes_Core_DOM type_wf_Core_DOM known_ptr_Core_DOM
    get_child_nodes_Core_DOM get_child_nodes_locs_Core_DOM
  for type_wf :: "(_) heap ⇒ bool"
  and known_ptr :: "(_) object_ptr ⇒ bool"
  and type_wf_Core_DOM :: "(_) heap ⇒ bool"
  and known_ptr_Core_DOM :: "(_) object_ptr ⇒ bool"
  and get_child_nodes :: "(_) object_ptr ⇒ (_, (_) node_ptr list) dom_prog"
  and get_child_nodes_locs :: "(_) object_ptr
    ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set"
  and get_child_nodes_Core_DOM :: "(_) object_ptr
    ⇒ (_, (_) node_ptr list) dom_prog"
  and get_child_nodes_locs_Core_DOM :: "(_) object_ptr
    ⇒ ((_) heap ⇒ (_) heap ⇒ bool) set" +
  assumes known_ptr_impl: "known_ptr = ShadowRootClass.known_ptr"
  assumes type_wf_impl: "type_wf = ShadowRootClass.type_wf"
  assumes get_child_nodes_impl: "get_child_nodes = a_get_child_nodes"
  assumes get_child_nodes_locs_impl:
    "get_child_nodes_locs = a_get_child_nodes_locs"
```

With this construction, lemmas such as `get_child_nodes_pure` can now be proven for our updated definition without having to consider all the implementations for elements and other classes. To do so, we leverage the splitter rule `CD.get_child_nodes_splits` from the previous universe to obtain two cases: one where we can assume a pointer from the previous universe, in which case we can immediately finish the proof by using the previous lemma, and one where we know that we have a pointer of our new type:

```
lemma get_child_nodes_pure [simp]:
  "pure (get_child_nodes ptr) h"
  unfolding get_child_nodes_def a_get_child_nodes_tups_def
proof(split CD.get_child_nodes_splits, rule conjI; clarify)
  assume "known_ptr_Core_DOM ptr"
  then show "pure (get_child_nodes_Core_DOM ptr) h"
    by simp
next
  assume "¬ known_ptr_Core_DOM ptr"
  then show "pure (invoke [(is_shadow_root_ptr_object_ptr,
      get_child_nodes_shadow_root_ptr ∘ the ∘ cast_object_ptr2shadow_root_ptr)] ptr ()) h"
    by(auto simp add: get_child_nodes_shadow_root_ptr_def
              intro: bind_pure_I split: invoke_splits)
qed
```

After interpreting the locale, we can obtain our new definitions and lemma as before, with the ones from the Core DOM universe being shadowed, but still being accessible by their fully-qualified name (e. g., `Core_DOM.get_child_nodes`).

4.3 Graph Relations

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 root sub-trees while the latter traverses (open) shadow roots prior to traversing its child nodes. While not part of the DOM API, it is an important concept that many other methods rely on, which is why we formalise it. We have seen this difference in Figure 4.3 already, where the code snipped was subject to the shadow-including tree order in the example that used shadow roots, which meant that, for example, the buttons on the bottom were skipped. We formalise this concept as as a **`partial_function`**, because it can potentially loop infinitely on a heap that contains cycles. Only in proofs containing the proper well-formedness assumptions can we actually use the definition then in a meaningful way. The definition looks 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 in the standard, it implicitly assumes that the algorithm computing a shadow-including tree order always terminates. In our formalisation, this is expressed as the property that "for all well-formed heaps, the (partial) function `to_tree_order_si` does not produce an error", i. e., non-termination is mapped to a value of our error type. Therefore, for any meaningful proof involving this function, we again use well-founded induction stemming from the acylicity of our combined `childNodes` and `shadowRoot` relations.

Instead of using `partial_function`, we could also use regular (total) functions to some extent. The main difference that would be required is the presence of a guard clause in the definition of the function itself to ensure the totality of the function. The guard clause would then check the well-formedness of the heap before proceeding with the actual computation, or would return an error value if the heap is not well-formed. While this might seem like a simple way to avoid using partial functions, which are generally trickier to handle, this approach would have the disadvantages of, first, cluttering the function definition with information that we consider to be more appropriate in the forms of proofs, second, merely shifting termination proofs from separate lemmas into Isabelle's function termination framework and, third, creating references from function definitions to our big well-formedness predicates, constraining the way in which we order definitions and lemmas in our proof document.

The model of our running example, shown in Figure 4.4, demonstrates the three different kinds of edges that we now have in our node tree: children, shadow roots, and disconnected nodes. In the next section, we will see updated well-formedness predicates that assure that all these relations form an acyclic graph, among other properties, some of which we have seen in the previous chapter. Using these updated predicates, we can show that, for example, `to_tree_order_si` always terminates on a well-formed heap.

4.4 Heap Invariants

As we have seen with the core DOM in Section 3.3, a well-formed heap needs to fulfill several properties that can either be modelled as type 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 formalisation, 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:

```
abbreviation "safe_shadow_root_element_types ≡ {''article'',
  ''aside'', ''blockquote'', ''body'', ''div'', ''footer'',
  ''h1'', ''h2'', ''h3'', ''h4'', ''h5'', ''h6'', ''header'',
  ''main'', ''nav'', ''p'', ''section'', ''span''}"

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

The constraint `shadow_root_valid` is described in the standard informally as the requirement that a DOM instance does not contain "disconnected" shadow roots. Unfortunately, many of these kind of invariants are only implicit in the standard, which makes it difficult to grasp what exactly are the allowed states of a heap. This invariant, along with the ones that follow, have all been extracted from the DOM specification and formalised as

soon as they were needed for proofs. This means we ended up with a small set of only the most critical invariants.

Another invariant 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
  ))"
```

We also would like to ensure that all elements only ever refer to shadow roots that actually exist in the heap:

```
definition all_ptrs_in_heap :: "(_) heap ⇒ bool"
where
  "all_ptrs_in_heap h = (∀host shadow_root_ptr.
    (h ⊢ get_shadow_root host →_r Some shadow_root_ptr) ⟶
      shadow_root_ptr |∈| shadow_root_ptr_kinds h)"
```

Finally, to ensure that any DOM instance 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, together with the `childNodes` relation, is acyclic.

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

```
definition heap_is_wellformed :: "(_) heap ⇒ bool"
  where
    "heap_is_wellformed h ⟷ CD.heap_is_wellformed h ∧
      acyclic (CD.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-formed heap requires that the regular parent-child relation *together* with the shadow root-host relation is acyclic, so we combine them and need `CD.parent_child_rel h ∪ host_shadow_root_rel h` to be acyclic.

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

```
lemma to_tree_order_si_ok:
  assumes "heap_is_wellformed h" and "known_ptrs h" and "type_wf h"
  assumes "ptr |∈| object_ptr_kinds h"
shows "h ⊢ ok (to_tree_order_si ptr)"
```

This lemma 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.

Additionally, we introduce a short-hand predicate `valid_heap` for the three predicates that occur in most lemmas. It captures `heap_is_wellformed`, ensures that the heap only contains pointers and objects whose types correspond (`type_wf`), and permits only "known" pointers (`known_ptrs`), a property related to the extensibility of the formal model:

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

Of course, we also show that all our DOM methods preserve the `valid_heap` predicate, i. e., if the predicate held before the method was called, it also holds on the modified heap returned by the method. All these proofs can be found in the full formalisation document.

4.5 Interfacing Shadow Trees: Slotting

The DOM standard describes a dedicated mechanism for allowing a restricted form of interaction between a shadow root and its host (and therefore also the tree containing the hosting `Element`). The interface mechanism allowing this kind of interaction is called *slotting*, which "virtually maps" nodes (*slotables*) from one sub-tree into *slots* of another sub-tree without actually re-attaching them to other nodes. We have already seen in Figure 4.1 how slotting works in practice and will now have a more detailed look.

A *slot* is an `Element` with a `slot` tag. Slots must only occur inside shadow trees and provide a container for other nodes. A slot has a `name` attribute, which can later be referred to in order to reference the slot. If a slot has no name, it becomes the default slot. If a slot does not receive any slotables during slotting, its children will be rendered instead.

A *slotable* is any `Node` that is a *direct* child of a shadow root host. Slotables can have a `slot` attribute (in the case of an `Element`) which specifies the target slot. If no such attribute is present, the slotable will be slotted into the default slot.

The two most important methods defined in the DOM standard for inspecting the slotting mechanism are getting the *assigned slot* and *assigned nodes*. The method `assigned_slot` takes a node, i. e., either an `Element` or `CharacterData`, and returns the

slot that it will be assigned to, if any. Additionally, it takes a boolean flag that indicates whether it should obey the `mode` of the shadow tree: if the shadow root is closed, `assigned_slot` will return `None`.

The formalisation of `assigned_slot` can be found in Listing 4.1. Until Line 12, we ensure that the given node does indeed have a parent with a shadow root, which we obtain. If the shadow tree is open, we search the shadow tree in tree-order (Line 15) for the correct slot; that is, the first element with a *slot* tag and a `name` attribute that matches the `slot` attribute of the given slotable. These checks are carried out from Line 26 onward.

The method for getting the assigned nodes, `assigned_nodes` in our formalisation, takes an `Element` (the slot) and returns the list of nodes that will later be rendered in place of its children. If the passed element is not a slot, it will return an error. We formalise `get_assigned_nodes` in Isabelle/HOL as follows:

```
definition assigned_nodes :: "(_) element_ptr ⇒ (_, (_) node_ptr list) dom_prog"
where
  "assigned_nodes slot = do {
    tag ← get_tag_name slot;
    (if tag ≠ ''slot''
    then error HierarchyRequestError
    else return ());
    root ← get_root_node (cast slot);
    if is_shadow_root_ptr_kind root
    then do {
      host ← get_host (the (cast root));
      children ← get_child_nodes (cast host);
      filter_M (λslotable. do {
        found_slot ← assigned_slot False slotable;
        return (found_slot = Some slot)}) children}
    else return []}"
```

Basically, the method definition works by going from `slot` to its root node `root`, then to `root`'s `host`, and finally iterating over `host`'s children and calling `assigned_slot` to find all nodes that would return the given slot `slot` as their assigned slot. This algorithm follows the specification from the DOM standard.

Since the definitions are relatively complex, we will now try to describe `assigned_slot` and `assigned_nodes` by using a few properties to show *what* they do rather than *how*. One useful property is that if one tries to invoke `assigned_nodes` on an element that is not a slot, one will get an error. Or, to put it the other way round, if `assigned_nodes` terminates without an error, the reference element was indeed a slot:

```
lemma assigned_nodes_slot_is_slot:
  assumes "h ⊢ ok (assigned_nodes slot)"
shows "h ⊢ get_tag_name slot →ᵣ ''slot''"
```

Listing 4.1: Formalisation of `assigned_slot`

```
definition assigned_slot :: "bool ⇒ (_) node_ptr
  ⇒ (_, (_) element_ptr option) dom_prog" where
"assigned_slot open_flag slotable = do {
  parent_opt ← get_parent slotable;
  (case parent_opt of
    Some parent ⇒
      if is_element_ptr_kind parent
      then do {
        shadow_root_ptr_opt ← get_shadow_root (the (cast parent));
        (case shadow_root_ptr_opt of
          Some shadow_root_ptr ⇒ do {
            shadow_root_mode ← get_mode shadow_root_ptr;
            if open_flag ∧ shadow_root_mode ≠ Open
            then return None
            else first_in_tree_order (cast shadow_root_ptr)
              (λptr. if is_element_ptr_kind ptr
                then do {
                  tag ← get_tag_name (the (cast ptr));
                  name_attr ← get_attribute (the (cast ptr))
                    ''name'';
                  slotable_name_attr ←
                    (if is_element_ptr_kind slotable
                    then get_attribute (the (cast slotable))
                      ''slot''
                    else return None);
                    (if (tag = ''slot''
                    ∧ (name_attr = slotable_name_attr
                      ∨ (name_attr = None
                        ∧ slotable_name_attr = Some '''')
                      ∨ (name_attr = Some ''''
                        ∧ slotable_name_attr = None)))
                  then return (Some (the (cast ptr)))
                  else return None)}
              else return None)}
          | None ⇒ return None)}
      else return None
| _ ⇒ return None)}"
```

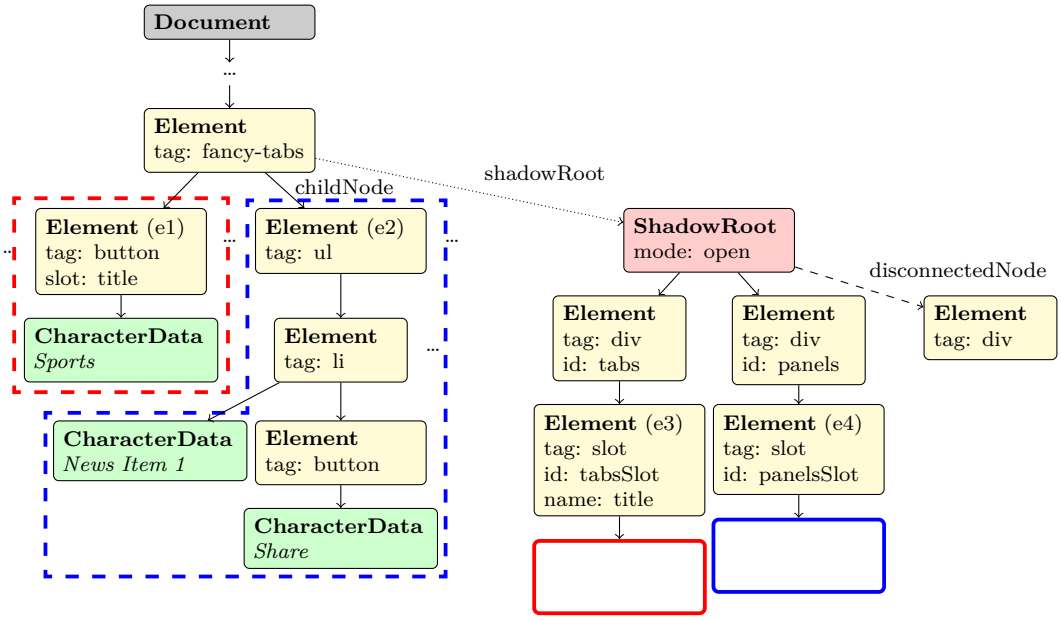


Figure 4.6: Visualisation of the nodes that get assigned to their slots during the slotting process. The sub-trees surrounded by the dashed boxes indicate the slotables that get displayed as if they were located in the solid boxes of the same colour. Elements $e1$ and $e2$ are the only assigned nodes for $e3$ and $e4$, respectively, and $e3$ and $e4$ are the assigned slots of $e1$ and $e2$. The slotting algorithm uses the `slot` and `name` attributes to calculate these matchings.

The proof is actually straight forward, as this property is checked by the specification of `assigned_nodes` within the first few lines. Therefore, we simply need to unroll the definition for the proof.

Another important property of `assigned_nodes` is the following: assigned nodes never overlap. To be more precise, we prove that if we have two calls to `assigned_nodes` given slots `slot` and `slot'` that return the assigned nodes `nodes` and `nodes'`, respectively, then these two lists will share no single pointer if the slots were different:

```
lemma assigned_nodes_different_ptr:
  assumes "h ⊢ assigned_nodes slot →ᵣ nodes"
  assumes "h ⊢ assigned_nodes slot' →ᵣ nodes'"
  assumes "slot ≠ slot'"
shows "set nodes ∩ set nodes' = {}"
```

In order to demonstrate the relationship between slots and their assigned nodes, we highlight the pairings in the tree figure of our running example in Figure 4.6. It is important to remember that the slotables never actually get moved into the children of their assigned slots. It is only during the flattening, which is part of the rendering process in a web browser, that these slotables get displayed in their assigned position.

4.6 Flattening Shadow Trees

Slots and slotting are a concept with subtle corner cases. To provide a semantics of shadow trees and slotting in terms of a DOM without shadow roots, we formalise a method `flatten_dom`. This method takes a node tree with slots and slotables and produces a *flattened* node tree without shadow trees. Roughly speaking, we formalise the process that lead us to Figure 4.2b: we combine Figure 4.1b with Figure 4.2a.

While flattening is *not* described in the DOM standard, we see two benefits in formalising it: first, it provides a semantics of shadow trees and slots in terms of a DOM without shadow trees and, second, by proving properties of the flattening algorithm we are strengthening the understanding of standard methods such as `assigned_nodes` and `assigned_slot`, which build the basis of our flattening definition. Moreover, the flattened version of a node tree can also be understood as a "rendering presentation" that could, theoretically, be used by a rendering engine that does not support shadow trees.

The formalisation of the flattening algorithm contains a number of `map` statements, which can perhaps be more easily understood by interpreting them as loops. The definition looks as follows:

```
definition flatten_dom :: "(_, unit) dom_prog" where
  "flatten_dom = do {
    tups ← element_ptr_kinds_M ≫= map_filter_M2 (λelement_ptr. do {
      tag ← get_tag_name element_ptr;
      nodes ← assigned_nodes element_ptr;
      (if tag = ''slot'' ∧ nodes ≠ []
      then return (Some (element_ptr, nodes))
      else return None)});
    forall_M (λ(slot, nodes). do {
      get_child_nodes (cast slot) ≫= forall_M remove;
      forall_M (append_child (cast slot)) nodes
    }) tups;
    shadow_root_ptr_kinds_M ≫= forall_M (λshadow_root_ptr. do {
      host ← get_host shadow_root_ptr;
      get_child_nodes (cast host) ≫= forall_M remove;
      get_child_nodes (cast shadow_root_ptr)
        ≫= forall_M (append_child (cast host));
      remove_shadow_root host
    });
    return ()}"
```

The definition can be grouped into three parts: first, we build a list of all slots along with the nodes that they will receive as returned by `assigned_nodes`. Second, we clear all children from the slots and then insert the assigned nodes, moving them into the place of the cleared children. Third, we remove all shadow roots after replacing the children of their hosts with the children of the shadow roots.

The main effect of this flattening algorithm is the fact that children appear to be in

a different place after slotting took place. We capture this observation in the following lemma which states that every element in a heap that has at least one assigned node before flattening, will have these nodes as children afterwards:

```
lemma flatten_dom_assigned_nodes_become_children:
  assumes "valid_heap h"
  assumes "h ⊢ flatten_dom →h h'"
  assumes "h ⊢ assigned_nodes slot →r nodes"
  assumes "nodes ≠ []"
  shows "h' ⊢ get_child_nodes (cast slot) →r nodes"
```

The proof idea stems from the line `forall_M (append_child (cast slot)) nodes` in the definition, which adds all assigned nodes to the now child-free slot.

Properties such as `flatten_dom_assigned_nodes_become_children` could be used to supplement the DOM standard to provide a precise characterisation of the flattening algorithm. They would serve as additional documentation for developers trying to understand the algorithm, but could also be used to derive concrete test cases that could then be evaluated on real DOM implementations. Coming up with more properties that would be useful for these purposes is a difficult task in general, however, similar to the challenge of coming up with useful test cases for regular software development.

Summary

In this chapter, we have seen how we can extend our formal model of the core DOM with shadow roots. Using our extension mechanism in Isabelle/HOL, we were able to do so in an iterative manner, which allowed us to stay as close to the official standard as possible while benefiting from a higher-order logic. In the process, we have gained new insights and a better understanding of the DOM with shadow roots in general, thus also being applicable to implementation of the standard in the wild. It is important to keep in mind that our main interest of shadow roots stems from the fact that they are the main *component mechanism* of the DOM standard. Now, we can start to explore *formally* how suitable shadow roots are for that purpose, and whether any changes or additions to the standard might be useful–all being backed by our model and the strong guarantees of using higher-order logic and proofs.

5 Compliance to the Standard

In the previous two chapters, we have built a formal model of the DOM and presented our approach of how to translate descriptions in semi-structured English into HOL functions. However, there remains the possibility that we introduced errors during this translation, which would limit the applicability of any proof and analysis results to the official standard. Therefore, before we continue with the introduction of web components into our model, we will seek ways to reduce the possibility of translation errors.

Thankfully, there is an approach that we can use to strengthen the connection between standard document written in prose and our formal document. The standardisation body not only provides the specification document, but also a comprehensive *test suite* to be used by implementations that wish to confirm their compliance to the standard. While the creation of such a suite has been a manual effort, too, it is being maintained by some of the people that are involved in the standard itself. In addition, this suite is also used by popular web browser implementations such as Mozilla Firefox [51] or Google Chrome [31] to confirm their compliance.

In this chapter, therefore, we will show that the formal model that we built in the previous two chapters is not simply inspired by the official DOM standard, but does indeed *pass all relevant test cases* from the official test suite which is also used by modern web browsers to show their compliance to the standard. After giving an overview of our approach, we, first, identify all relevant test cases, which are written in HTML and JavaScript. Since most of them follow a relatively simple pattern, we can, second, employ a straight-forward, semi-automatic translation from these test cases into HOL predicates, so we can then use symbolic execution to show that our formalisation passes these test cases. We consider a test successfully passed if we can prove that the corresponding HOL predicate holds, given the concrete heap from the test case. Finally, we give an outlook of how this infrastructure can be used in turn to enhance both *fDOM* and the official specification document. All Isabelle lemmas in this chapter can be found online as part of the Core DOM [16] or Shadow DOM [14] entry.

5.1 Formal Software Standards

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. 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.

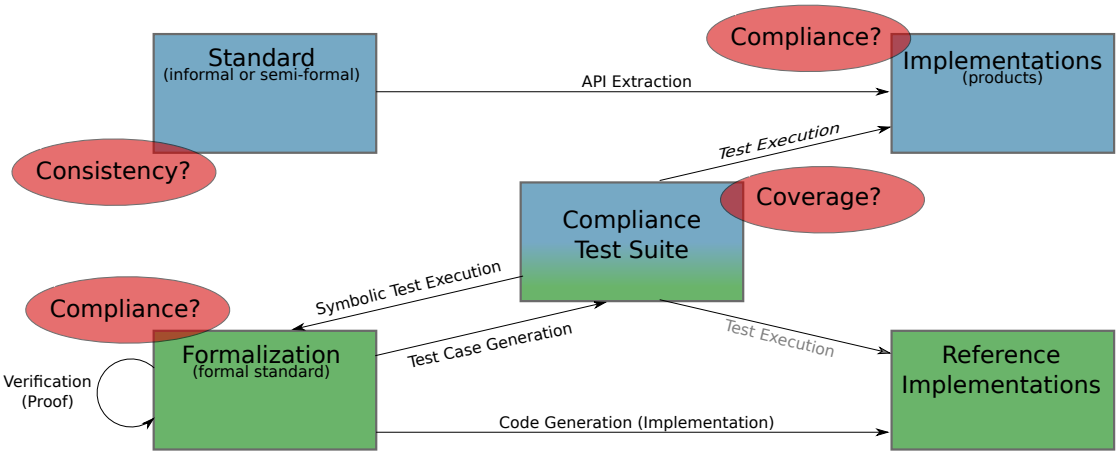


Figure 5.1: Using test and proof for establishing strong links between formal standards, compliance test suites, and implementations.

Even when such a formalisation 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 section, we present an approach that brings all three involved artefacts–the (semi-)formal standard, the formalisation of the standard, and the implementation–closer together by combining verification, symbolic execution, and specification-based testing. We will then apply this approach to show the compliance of *fDOM* with shadow roots to the official DOM specification.

Figure 5.1 illustrates the overall scenario for both traditional development of standards (upper part) and the integration of "test and proof"-activities (lower part).

First, let us recall the process and challenges of developing informal or semi-formal standards and implementations and their implementations: 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 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 of automatically extracting 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 artefact 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 formalisation (e. g., expressed in Isabelle/HOL; a formalisation that we can extract code from) of the standard exists. Of course, if we start with an informal standard, the question arises to which extent the formalisation is a faithful representation of the informal (or semi-formal) standard, i. e., the *compliance* of the formalisation. 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 [20]), we could 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.

In the following, however, we will focus on the "Symbolic Test Execution" arrow from Figure 5.1 to strengthen the trust in the compliance of *fDOM* to the standard.

5.2 Selecting Test Cases

The official test suite of the of the W3C, "web-platform-test" [68], covers a variety of features relevant for W3C standards. It contains over 30 000 HTML files, which all represent one *test* of some group of features or API of a standard and are not necessarily limited to the DOM. Each test, then, is divided into a number of *sub-tests*, which can range from just a few to tens of thousands per test, each one consisting of a few lines of JavaScript testing one specific feature or condition as specified in the standard. Relevant for us are the `dom` and `shadow_dom` test groups, which contain 262 and 123 HTML files, respectively. Even those, however, cover large parts of the DOM and also HTML specifications, much more than supported in *fDOM*. Recall that the *fDOM* focuses on the node-tree and methods that create, inspect, or modify its nodes. Examples of unsupported parts include *events*, e. g., executing JavaScript when the user clicks on a Button, or certain data classes such as `NodeList`, which is a live collection interface of the DOM, modeled in the *fDOM* as ordinary HOL lists, meaning that test cases from the official suite are not applicable. The full list of relevant tests can be found in Table 5.1, which we have all translated; the tests that we deemed irrelevant according to above criteria can be found in Table 5.2.

Each test is divided into sub-tests, of which not every single one is necessarily relevant or expressible in *fDOM*. Examples of such tests are ones that test each and every step of an algorithm, which is relevant, but might also check that a certain attribute related to an unsupported feature is present, which is not relevant. We therefore deemed a test

Table 5.1: All test cases from the official suite that are relevant have been translated. Since the *fDOM* focuses on the node-tree and related methods, most of the translated test cases are concerned with these methods.

Test Name	**Relevance**
/dom/interface-objects.html	relevant
/dom/nodes/CharacterData-appendChild.html	relevant
/dom/nodes/CharacterData-appendData.html	relevant
/dom/nodes/CharacterData-data.html	relevant
/dom/nodes/CharacterData-deleteData.html	relevant
/dom/nodes/CharacterData-insertData.html	relevant
/dom/nodes/CharacterData-remove.html	relevant
/dom/nodes/CharacterData-replaceData.html	relevant
/dom/nodes/CharacterData-substringData.html	relevant
/dom/nodes/ChildNode-after.html	relevant
/dom/nodes/ChildNode-before.html	relevant
/dom/nodes/ChildNode-replaceWith.html	relevant
/dom/nodes/Document-adoptNode.html	relevant
/dom/nodes/Document-constructor.html	relevant
/dom/nodes/Document-createElement.html	relevant
/dom/nodes/Document-doctype.html	relevant
/dom/nodes/Document-getElementById.html	relevant
/dom/nodes/Document-getElementsByTagName.html	relevant
/dom/nodes/Element-childElement-null.html	relevant
/dom/nodes/Element-childElementCount-dynamic-add.html	relevant
/dom/nodes/Element-childElementCount-dynamic-remove.html	relevant
/dom/nodes/Element-childElementCount-nochild.html	relevant
/dom/nodes/Element-childElementCount.html	relevant
/dom/nodes/Element-firstElementChild.html	relevant
/dom/nodes/Element-getElementsByClassName.html	relevant
/dom/nodes/Element-getElementsByTagName.html	relevant
/dom/nodes/Element-lastElementChild.html	relevant
/dom/nodes/Element-nextElementSibling.html	relevant
/dom/nodes/Element-previousElementSibling.html	relevant
/dom/nodes/Element-remove.html	relevant
/dom/nodes/Element-siblingElement-null.html	relevant
/dom/nodes/Element-tagName.html	relevant
/dom/nodes/Node-appendChild.html	relevant
/dom/nodes/Node-childNodes.html	relevant
/dom/nodes/Node-contains.html	relevant
/dom/nodes/Node-insertBefore.html	relevant
/dom/nodes/Node-isConnected.html	relevant
/dom/nodes/Node-parentElement.html	relevant
/dom/nodes/Node-parentNode.html	relevant
/dom/nodes/Node-removeChild.html	relevant
/dom/nodes/Node-replaceChild.html	relevant
/dom/nodes/Node-textContent.html	relevant
/dom/nodes/ParentNode-append.html	relevant
/dom/nodes/ParentNode-prepend.html	relevant
/dom/nodes/append-on-Document.html	relevant
/dom/nodes/insert-adjacent.html	relevant
/dom/nodes/prepend-on-Document.html	relevant
/dom/nodes/remove-and-adopt-crash.html	relevant
/dom/nodes/rootNode.html	relevant

Table 5.2: Test cases from the official suite that are *not* relevant to our formalisation, either due to covering DOM features that we do not support or JavaScript specific issues.

Test Name	Relevance
/dom/(events|traversal|ranges|lists|collections|abort)/.*	irrelevant
/dom/historical.html	irrelevant
/dom/interfaces.html?exclude=Node	irrelevant
/dom/nodes/(Text|ProcessingInstruction|DocumentType|DOMImplementation).*	irrelevant
/dom/nodes/.*(xhtml|xvg|xml|svg)	irrelevant
/dom/nodes/CharacterData-surrogates.html	irrelevant
/dom/nodes/Comment-constructor.html	irrelevant
/dom/nodes/Document-URL.html	irrelevant
/dom/nodes/Document-characterSet-normalization.html	irrelevant
/dom/nodes/Document-contentType/.*	irrelevant
/dom/nodes/Document-createAttribute.html	irrelevant
/dom/nodes/Document-createComment.html	irrelevant
/dom/nodes/Document-createElement-namespace.html	irrelevant
/dom/nodes/Document-createElementNS.html	irrelevant
/dom/nodes/Document-createProcessingInstruction.html	irrelevant
/dom/nodes/Document-createTextNode.html	irrelevant
/dom/nodes/Document-createTreeWalker.html	irrelevant
/dom/nodes/Document-getElementsByClassName.html	irrelevant
/dom/nodes/Document-getElementsByTagNameNS.html	irrelevant
/dom/nodes/Document-implementation.html	irrelevant
/dom/nodes/Document-importNode.html	irrelevant
/dom/nodes/Element-children.html	irrelevant
/dom/nodes/Element-classlist.html	irrelevant
/dom/nodes/Element-closest.html	irrelevant
/dom/nodes/Element-firstElementChild-namespace.html	irrelevant
/dom/nodes/Element-getElementsByTagName-change-document-HTMLNess.html	irrelevant
/dom/nodes/Element-getElementsByTagNameNS.html	irrelevant
/dom/nodes/Element-hasAttributes.html	irrelevant
/dom/nodes/Element-insertAdjacentElement.html	irrelevant
/dom/nodes/Element-insertAdjacentText.html	irrelevant
/dom/nodes/Element-matches.html	irrelevant
/dom/nodes/Element-removeAttributeNS.html	irrelevant
/dom/nodes/Element-webkitMatchesSelector.html	irrelevant
/dom/nodes/MutationObserver.*	irrelevant
/dom/nodes/Node-baseURI.html	irrelevant
/dom/nodes/Node-cloneNode.html	irrelevant
/dom/nodes/Node-compareDocumentPosition.html	irrelevant
/dom/nodes/Node-constants.html	irrelevant
/dom/nodes/Node-isEqualNode.html	irrelevant
/dom/nodes/Node-isSameNode.html	irrelevant
/dom/nodes/Node-lookupNamespaceURI.html	irrelevant
/dom/nodes/Node-nodeName.html	irrelevant
/dom/nodes/Node-nodeValue.html	irrelevant
/dom/nodes/Node-normalize.html	irrelevant
/dom/nodes/Node-properties.html	irrelevant
/dom/nodes/NodeList-Iterable.html	irrelevant
/dom/nodes/ParentNode-children.html	irrelevant
/dom/nodes/ParentNode-querySelector-All-xht.xht	irrelevant
/dom/nodes/ParentNode-querySelector-All.html	irrelevant
/dom/nodes/attributes.html	irrelevant
/dom/nodes/case.html	irrelevant
/dom/nodes/getElementsByClassName.*	irrelevant
/dom/nodes/query-target-in-load-event.html	irrelevant
/dom/nodes/remove-unscopable.html	irrelevant

relevant if the majority or core of the test applies to *fDOM*. Nevertheless, we are able to capture all relevant functional tests for the *fDOM*.

Another reason for exclusion is the fact that we cannot easily utilize test cases regarding type checks, as we decided to formalise a strongly typed model. The official compliance test suite contains many typing-related tests, mainly because of one of 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 formalise.

5.3 Translating Test Cases

The W3C test cases are written in JavaScript, which is embedded into the DOM instance under test. Most of them follow the same pattern: each sub-test consists of a call to a method `test` from the test harness, which takes a single function that queries and modifies the current DOM instance. Additionally, the test function calls various `assert` functions, that–if the check fails–throw an exception and cause the whole sub-test to fail. If the function reaches the end, the test is deemed successful. Figure 5.2 contains an example of such a sub-test for the function `insertBefore` and its translation into HOL. It checks whether the DOM method throws a certain exception if called with a certain combination of arguments. We formalised this test into our state-exception-monad and use a predicate `test` to convert the program to an Isabelle proof goal, with the proof obligation of showing the absence of exceptions.

It is important to note that we do not employ a full-featured JavaScript-to-HOL translator, which would be very complex; instead, we limit ourselves to this semi-automatic approach which is able to detect a number of patterns used by the test cases and maps them to pre-defined *fDOM* and state-monad constructs. The converter that we built is written in JavaScript itself and uses a JavaScript parsing library to parse the abstract syntax trees (AST) of the test cases. We then iterate the AST to detect constructs such as calls to the `test()` method, variables assignments, or method calls, which we then translate into methods of our HOL test framework, such as the `do` syntax for monads. For the construction of the heap, we simply use an HTML parser to parse the HTML file containing the test case and then a tree walker on the node tree to output our HOL finite map that represents the heap.

For the actual Isabelle proof showing that the given program does not result in an error, we have multiple possible options. The key difference between these test cases

```
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.')
```

(a) Sub-test from the W3C test suite, written in JavaScript.

```
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.*)
```

(b) The same sub-test translated into HOL for *fDOM*.

Figure 5.2: Translating a sub-test from the W3C test suite into HOL.

and our "regular" lemmas is that the translated test cases all start with a concrete heap (`Node_insertBefore_heap` in Figure 5.2b), derived from the HTML document that contained the original sub-test, which leaves us with *no free variables* in our proof goal. Therefore, the traditional approach of using the simplifier with a few definitions and additional lemmas is possible, but tedious. Instead, we employ symbolic evaluation in Isabelle as the tactic `code_simp`, which uses the simplifier internally together with a set of pre-defined equations that are suitable for symbolic execution, and `eval`, which generates and executes ML code for the proof. More detailed information about the proof tactics and their configuration can be found in the Isabelle manual [70] or the code generator documentation [34]. In general, `code_simp` is slower, but more trustworthy than using `eval`, since the latter adds the whole code generator setup to the amount of code that is outside of the trust guarantees of Isabelle. Table 5.3 shows the run-times for our core DOM test cases and Table 5.4 for the tests related to shadow roots, which all show that `eval` is orders of magnitudes faster than `code_simp`. All benchmarks have been run on a single core of a Intel Core i7-6600U CPU @ 2.60GHz of a laptop and have been run three times, of which we took the median value for each test. We have translated all sub-tests that we deemed relevant, and we have proven all translated sub-tests to terminate without error, meaning that *fDOM* passes all relevant test cases.

5.4 Beyond Compliance: Enhancing the Specification

The test suite is an important part of showing the compliance of any implementation. Now that we have established reasonable confidence in the compliance of our model, we can actually use our formalisation to also contribute back, towards *fDOM*, the test suite, and also the underlying specification itself. Tests are a very limited way of showing important properties, as they only do so for concrete input values, such as a simple DOM instance. Since we do not have this kind of limitation in Isabelle, we can–inspired by the test cases–generalise them to generic theorems that show the corresponding property for

Table 5.3: Execution times of all translated test cases of the core DOM in seconds. The Description is directly taken from the test cases.

Method	**Description**	t_{eval}	t_{code_simp}
adoptNode	Adopting an Element called 'x<' should work.	4.88	67.74
adoptNode	Adopting an Element called ':good:times:' should work.	2.27	42.97
getElementById	Document.getElementById with a script-inserted element	2.57	69.93
getElementById	update 'id' attribute via setAttribute/removeAttribute	2.82	63.31
getElementById	Ensure that the id attribute only affects elements present in a document	2.45	51.97
getElementById	in tree order, within the context object's tree	2.64	83.99
getElementById	Modern browsers optimize this method with using internal id cache. This test checks that their optimization should effect only append to 'Document', not append to 'Node'.	2.54	32.12
getElementById	changing attribute's value via 'Attr' gotten from 'Element.attribute'.	2.67	59.69
getElementById	update 'id' attribute via element.id	2.57	59.81
getElementById	where insertion order and tree order don't match	2.82	466.99
getElementById	Inserting an id by inserting its parent node	2.45	73.97
getElementById	Document.getElementById must not return nodes not present in document	2.47	81.37
insertBefore	Calling insertBefore an a leaf node Text must throw HIERARCHY_REQUEST_ERR.	2.22	0.95
insertBefore	Calling insertBefore with an inclusive ancestor of the context object must throw HIERARCHY_REQUEST_ERR.	4.93	10.78
insertBefore	Calling insertBefore with a reference child whose parent is not the context node must throw a NotFoundError.	2.20	3.75
insertBefore	If the context node is a document, inserting a document or text node should throw a HierarchyRequestError.	2.66	46.68
insertBefore	Inserting a node before itself should not move the node	2.14	32.00
removeChild	Passing a detached Element to removeChild should not affect it.	2.05	4.70
removeChild	Passing a non-detached Element to removeChild should not affect it.	2.27	12.27
removeChild	Calling removeChild on an Element with no children should throw NOT_FOUND_ERR.	2.26	13.29
removeChild	Passing a detached Element to removeChild should not affect it.	2.36	26.61
removeChild	Passing a non-detached Element to removeChild should not affect it.	2.41	39.59
removeChild	Calling removeChild on an Element with no children should throw NOT_FOUND_ERR.	2.44	42.26
removeChild	Passing a value that is not a Node reference to removeChild should throw TypeError.	1.73	1.56

Table 5.4: Execution times of all translated test cases of shadow roots in seconds. Tests were terminated after a maximum of 10 minutes. The Description is directly taken from the test cases.

Method	**Description**	t_{eval}	t_{code_simp}
slots	'Slots: Basic.'	6.31	>600.00
slots	'Slots: Basic, elements only.'	6.26	>600.00
slots	'Slots: Slots in closed.'	6.79	>600.00
slots	'Slots: Slots in closed, elements only.'	6.33	>600.00
slots	'Slots: Slots not in a shadow tree.'	6.06	>600.00
slots	'Slots: Slots not in a shadow tree, elements only.'	9.13	>600.00
slots	'Slots: Distributed nodes for Slots not in a shadow tree.'	6.49	>600.00
slots	'Slots: Name matching'	5.88	>600.00
slots	'Slots: No direct host child.'	9.45	>600.00
slots	'Slots: Default Slot.'	6.23	>600.00
slots	'Slots: Slot in Slot does not matter in assignment.'	9.46	>600.00
slots	'Slots: Slot is assigned to another slot'	11.53	>600.00
slots	'Slots: Open > Closed.'	9.19	>600.00
slots	'Slots: Closed > Closed.'	6.82	>600.00
slots	'Slots: Closed > Open.'	8.89	>600.00
slots	'Slots: Complex case: Basi line.'	9.22	>600.00
slots	'Slots: Mutation: appendChild.'	13.95	>600.00
slots	'Slots: Mutation: Change slot= attribute 1.'	7.15	>600.00
slots	'Slots: Mutation: Change slot= attribute 2.'	12.28	>600.00
slots	'Slots: Mutation: Change slot= attribute 3.'	6.35	>600.00
slots	'Slots: Mutation: Remove a child.'	9.47	>600.00
slots	'Slots: Mutation: Add a slot: after.'	7.32	>600.00
slots	'Slots: Mutation: Add a slot: before.'	13.56	>600.00
slots	'Slots: Mutation: Remove a slot.'	7.02	>600.00
slots	'Slots: Mutation: Change slot name= attribute.'	6.68	>600.00
slots	'Slots: Mutation: Change slot slot= attribute.'	6.66	>600.00
slots_fallback	'Slots fallback: Basic.'	2.80	243.71
slots_fallback	'Slots fallback: Basic, elements only.'	2.81	202.66
slots_fallback	'Slots fallback: Slots in Slots.'	2.89	>600.00
slots_fallback	'Slots fallback: Slots in Slots, elements only.'	2.78	>600.00
slots_fallback	'Slots fallback: Fallback contents should not be used if a node is assigned.'	3.03	>600.00
slots_fallback	'Slots fallback: Slots in Slots: Assigned nodes should be used as fallback contents of another slot'	2.90	>600.00
slots_fallback	'Slots fallback: Complex case.'	3.04	>600.00
slots_fallback	'Slots fallback: Complex case, elements only.'	3.25	>600.00
slots_fallback	'Slots fallback: Mutation. Append fallback contents.'	3.81	>600.00
slots_fallback	'Slots fallback: Mutation. Remove fallback contents.'	3.44	>600.00
slots_fallback	'Slots fallback: Mutation. Assign a node to a slot so that fallback contens are no longer used.'	3.83	>600.00
slots_fallback	'Slots fallback: Mutation. Remove an assigned node from a slot so that fallback contens will be used.'	3.44	>600.00
slots_fallback	'Slots fallback: Mutation. Remove a slot which is a fallback content of another slot.'	3.69	>600.00

all possible inputs. For our example property, we generalize the test as follows:

```
lemma insert_before_ok:
  assumes "valid_heap h"
  assumes "parent |∈| object_ptr_kinds h"
  assumes "node |∈| node_ptr_kinds h"
  assumes "¬is_character_data_ptr_kind parent"
  assumes "cast node ∉ set |h ⊢ get_ancestors parent|_r"
  assumes "h ⊢ get_parent ref →_r Some parent"
  assumes "is_document_ptr parent ⟹ h ⊢ get_child_nodes parent →_r []"
  assumes "is_document_ptr parent ⟹ ¬is_character_data_ptr_kind node"
shows "h ⊢ ok (insert_before parent node (Some ref))"
```

Instead of creating three concrete elements, we can quantify over all possible elements. The assumptions give additional insight into the behaviour of `insert_before`; for example, 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.

Many kinds of test cases could be generalized in this manner. In general, we start by using an unspecified heap h instead of the concrete heap from the DOM under test–most likely, this change will lead to the failure of the test case. Now we look closer at the condition that lead to the failure of the test case and begin to add them to the assumptions of the lemma until the test case lemma can be proven correct again. This change will hopefully give a clearer picture of the necessary preconditions of the test than a concrete heap. In the same way, parts of the test case itself, e. g. the creation of a certain element, could be replaced as well. For each such replacement, the effect on the clarity of the whole test case has to be considered, since sometimes the use of a single DOM method can be more descriptive than trying to find a characterization in higher-order logic.

In order to assess which DOM methods in the standard would benefit the most from such generalized test cases, we assessed the state of the test suite with regards to their complexity and coverage, which is shown in Table 5.5. Some methods, such as `insertBefore`, are rather complex, i. e., they involve modifications to various nodes in the tree and feature many checks, but unfortunately only have little test coverage. Such methods are in need of a more complete test suite, which could be achieved by creating such generalized test cases and then using other techniques, e. g. test case generation, to complement existing tests. We only used rough, manual assessments of complexity and coverage, which can certainly be improved with more sophisticated methods, but our general approach remains the same. 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 formalisation. 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.

In many cases, the methods defined in the DOM standard need to fulfil important properties. These properties are neither spelled out explicitly nor does the compliance test suite contain test cases for them. During the formalisation of the standard, these

Table 5.5: The number of sub-tests regarding our supported DOM methods that are available from the official suite and relevant for us. 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. A function with high complexity but low coverage would therefore profit substantially.

	# Sub-Test 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	–

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 "valid_heap h"
  assumes "h ⊢ insert_before ptr new_child child_opt →ₕ h'"
  assumes "h' ⊢ get_child_nodes ptr →ᵣ children"
shows "⋀ptr' children'.
  h' ⊢ get_child_nodes ptr' →ᵣ children' ⟹ distinct children'"
```

Intuitively, 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 [20], the proven lemmas could be systematically turned into additional compliance test cases that ensure that actual implementations fulfil these crucial properties.

Summary

In this chapter, we have put our DOM formalisation into the bigger picture. We have increased the confidence in our model that it is a faithful representation of the standard. In addition, we have presented multiple other benefits of combining the *fDOM* with the compliance test suite, even though we leave most of them for future work; we can generalize test cases to HOL theorems, which inspires new theorems that might not necessarily be useful for bigger proofs, but can nevertheless provide properties that the test suite author deemed useful. These generalized theorems can then, together with other theorems, be used to generate more test cases, thus improving the compliance test suite and therefore also other implementations of web browsers.

6 A New Notion of Web Components

In the previous chapters, we have built a formal model of the DOM that complies to the standard and contains shadow roots as a technical basis for separating sub-trees from each other. However, as we will see in this chapter, if we take a step back and consider their use case more carefully, we will see that they fall short of this expectation of providing a modern component mechanism for several reasons.

Both producers and consumers of shadow root sub-trees are faced with the question: what guarantees come with these sub-trees, and how are these guarantees affected by the used DOM API methods? The DOM standard, as it stands, answers neither of these questions satisfactorily. In this chapter, therefore, we will 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 and consumers of components. We will start by formally defining *DOM-Components* and what it means for DOM methods to respect them, a concept which we will call DOM-Component *safety*. Then, we will classify the most important DOM methods according to that definition of safety, and we will finally see that shadow trees–in its current form–provide a quite unsatisfactory situation. All Isabelle definitions and proofs of this chapter can be found in [11].

6.1 A Formal Definition of Web Components

Many DOM methods, e. g., `get_element_by_id` and `get_root_node`, traverse the node tree exclusively along the `childNodes` relation (i. e., in shadow-excluding tree order or bottom-up, respectively). These methods will not traverse the DOM along the `shadowRoot` 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` relation.

6.1.1 Definition

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

Definition 6.1.1 (DOM-Component)**.** *A* DOM-Component *of an object pointer ptr is defined as the list of all objects in* (shadow-excluding) tree order *that are reachable from the root node of ptr.*

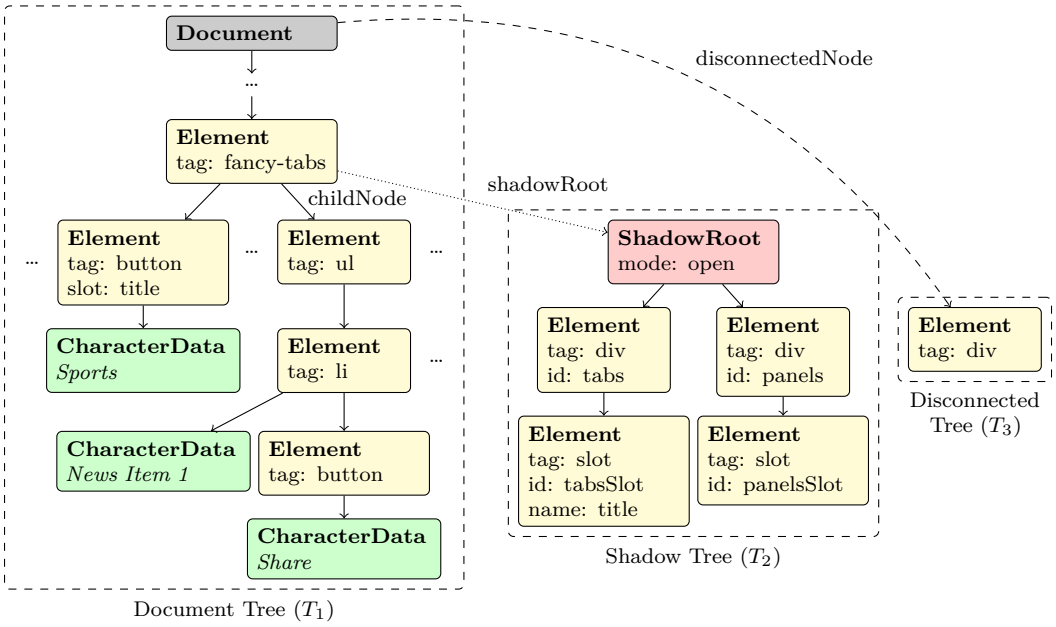


Figure 6.1: Running example from Figure 4.1 including a visualisation of DOM-Components.

In Isabelle, we define the following function for this definition of components:

```
definition get_dom_component :: "(_) object_ptr ⇒
  (_, (_) object_ptr list) dom_prog"
where
  "get_dom_component ptr = do {
    root ← get_root_node ptr;
    to_tree_order root
  }"
```

In other words, an object pointer *ptr* belongs to a DOM-Component T if and only if *ptr* is in the list of nodes that are reachable from the root of T via the `childNodes` relation. We show our updated running example in Figure 6.1), where we show how our component definition divides all pointers intro three separate trees T_1, T_2, and T_3.

6.1.2 Different Kinds of DOM-Components

In the example, we can see that not all components are of the same kind–our component definition naturally allows distinguishing three different types of components, based on the type of their root node:

Definition 6.1.2 (Document DOM-Component)**.** *A* Document DOM-Component *is a DOM-Component whose root node is of type* `Document`.

In Isabelle, we define this as follows:

```
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 6.1.3 (Shadow Root DOM-Component)**.** *A* Shadow Root DOM-Component *is a web component whose root node is of type* `ShadowRoot`*.*

```
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 and slotables to interact with the outer component.

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

Definition 6.1.4 (Disconnected DOM-Component)**.** *A* Disconnected DOM-Component *is a web component whose root node is of type* `Node`*.*

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

Disconnected components are not part of the main node tree and thus will not take part in the rendering of the final node tree. Usually, disconnected components are freshly created object graphs, e. g. by using `create_element`, that will become a part of a "regular" DOM instance by passing them as argument to methods such as `append_child`. On the other hand, all nodes that are serialized into an HTML page are *not* disconnected, because the successor relationship of this XML-based tree structure directly corresponds to the `childNodes` relationship of the DOM tree, and since the topmost element of the page becomes the `documentElement` of the main document, they are all connected.

6.1.3 Properties of DOM-Components

In the following, we will "test" our new definition of DOM-Components by proving some properties about them to convince ourselves that they behave as expected.

Let us start with the property that any arbitrary, but fixed pointer is actually inside its own component:

```
lemma get_dom_component_ptr:
  assumes "valid_heap h"
  assumes "h ⊢ get_dom_component ptr →r c"
shows "ptr ∈ set c"
```

The proof is less obvious than it might first seem. Recall that in order to calculate the component for a given object pointer `ptr`, we first obtain its root and then iterate downwards again–the lemma shown states that we will eventually reach our pointer `ptr` again. The proof skeleton looks as follows:

```
1  proof(insert assms(1) assms(4), induct ptr rule: heap_wellformed_induct_rev)
2    case (step child)
3    then show ?case
4    proof (cases "is_node_ptr_kind child")
5      case True
6      obtain node_ptr where node_ptr: "cast_node_ptr2object_ptr node_ptr = child"
7      ...
8      then obtain parent_opt where
9        parent: "h ⊢ get_parent node_ptr →_r parent_opt"
10     ...
11     then show ?thesis
12     proof (induct parent_opt)
13       case None
14       ...
15       then show ?case
16       ...
17     next
18       case (Some parent_ptr)
19       ...
20       then have "parent_ptr ∈ set c"
21       ...
22       show ?case
23       ...
24     qed
25   next
26     case False
27     then show ?thesis
28     ..
29   qed
30 qed
```

We start the proof by initiating an induction bottom-up inside a well-formed node tree. This means that we can assume if `child` has a parent and the component of the parent is `c`, then we know that the parent is inside that component. Now, given that `c` is also the component of `child`, we need to show that also `child` is inside the component. There are a few edge cases to consider, with the main proof path leading into Line 20, where we establish that the parent is indeed in `c`. Finally, we can use a lemma about `to_tree_order` that says that if a parent is in any given result from `to_tree_order`, its children will also be.

Another important fact, which will later be useful to prove that `get_child_nodes` is

strongly DOM-Component safe, is that for any DOM-Component `c`, any pointer is in `c` if and only if its children are in the same DOM-Component:

```
lemma get_dom_component_get_child_nodes:
  assumes "valid_heap h"
  assumes "h ⊢ get_component ptr →r c"
  assumes "h ⊢ get_child_nodes ptr' →r children"
  assumes "child ∈ set children"
shows "cast child ∈ set c ⟷ ptr' ∈ set c"
```

Perhaps most interestingly, it is impossible to have two different DOM-Components that overlap–any two DOM-Components either share all their nodes or none at all:

```
lemma get_dom_component_no_overlap:
  assumes "valid_heap h"
  assumes "h ⊢ get_dom_component ptr →r c"
  assumes "h ⊢ get_dom_component ptr' →r c'"
shows "set c ∩ set c' = {} ∨ c = c'"
```

This important observation allows us to partition any node tree, such as our example in Figure 6.1, into separate components without getting intro trouble.

6.2 Definition of DOM-Component Safety

Ultimately, web components should provide a certain form of safety guarantee to both component developers and consumers. Neither should a component unintentionally modify the consuming web application nor vice versa.

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

1. only operate within the DOM-Components of their arguments, as one could argue that it is expected that the method operates within their proximity (i. e., their DOM-Component). We will call these methods *strongly DOM-Component-safe*.
2. only operate within the DOM-Components of their arguments, in addition to any newly created DOM-Components. While these methods operate outside their perceived boundaries, they at least leave other, existing DOM-Components untouched. We will call these methods *weakly DOM-Component-safe*.
3. operate on arbitrary parts of a DOM instance. We will call these methods *unsafe*.

It would theoretically be possible to also use other measures of "proximity", for example, one parent- or child-step away instead of anywhere within the DOM-Component, or perhaps even a whole DOM heap way. However, using units of DOM-Components has the advantage that they are naturally divided by boundaries such as the `host-shadowRoot` boundary, which is also meant to separate code coming from different developers and thus might have different levels of trust.

6.2.1 Strong DOM-Component Safety

For strong DOM-Component safety, we define the following predicate in Isabelle:

```
definition is_strongly_dom_component_safe :: "(_) object_ptr set ⇒
  (_) object_ptr set ⇒ (_) heap ⇒ (_) heap ⇒ bool"
where
  "is_strongly_dom_component_safe S_arg S_result h h' = (
    let removed_pointers =
      fset (object_ptr_kinds h) − fset (object_ptr_kinds h') in
    let added_pointers =
      fset (object_ptr_kinds h') − fset (object_ptr_kinds h) in
    let arg_components =
      ⋃ptr ∈ (⋃ptr ∈ S_arg. set |h ⊢ get_dom_component ptr|_r) ∩
      fset (object_ptr_kinds h).  set |h ⊢ get_dom_component ptr|_r) in
    let arg_components' =
      (⋃ptr ∈ (⋃ptr ∈ S_arg. set |h ⊢ get_dom_component ptr|_r) ∩
      fset (object_ptr_kinds h').  set |h' ⊢ get_dom_component ptr|_r) in
    removed_pointers ⊆ arg_components ∧
    added_pointers ⊆ arg_components' ∧
    S_result ⊆ arg_components' ∧
    (∀outside_ptr ∈ fset (object_ptr_kinds h) ∩ fset (object_ptr_kinds h')
      − (⋃ptr ∈ S_arg. set |h  ⊢ get_dom_component ptr|_r).
      preserved (get_M outside_ptr id) h h'))"
```

Before we have a look into the details of this definition, let us consider an example first. To show that `get_child_nodes` is strongly DOM-Component safe, we prove the following lemma:

```
lemma get_child_nodes_strongly_dom_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_dom_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, for example, 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, which will make the proof much easier.

The predicate `is_strongly_dom_component_safe` then works as follows. First, we build four sets of interesting pointers: `removed_pointers` and `added_pointers` refer to the pointers that have been removed from and added to the heap `h'` (compared to heap `h`), respectively, followed by `arg_components` and `arg_components'`, which contain all pointers that are in any of the components of the "argument pointers" S_{arg}. The normal and

primed version of the set capture the components in `h` and `h'`, respectively, which can be quite different. For example, when `append_child` moves one of its arguments from one part of the DOM tree to another, that argument can switch components, which leads to very different `arg_components` and `arg_components'`. Second, we use these pointer sets to require that the following four statements hold: 1. All pointers that get removed were part of the argument pointers' components in the original heap, 2. all pointers that get added will be part of the argument pointers' components in the new heap, 3. also, all pointers that the function returns (S_{result}) must be part of the argument pointers' components in the new heap, and finally, 4. all pointers that have been in either heap and are *not* part of the argument pointers' components must remain unmodified (`preserved`).

Note that a method that would somehow have all pointers in their arguments and thus in S_{arg} would be trivially strongly DOM-Component safe. However, we do not consider this to be a problem for the usefulness of our safety definition, because if the developer is able to gather all available pointers before the call to this hypothetical method, then he has already been able to access all these parts of the node tree anyway.

6.2.2 Weak DOM-Component Safety

One class of DOM methods is inherently *not* strongly DOM-Component safe: constructors. All four constructors create a new component that is unrelated (using our definition of DOM-Component) to the argument of the constructor, as they will create a new disconnected node, a new, separate document, or a new shadow root, respectively, which all breaks our definition of strong safety. While this has potential for unexpected behaviour, we consider it "not quite as bad" as arbitrarily modifying the DOM instance. Formally, the definition is very close to the one of strong DOM-Component safety:

```
definition is_weakly_dom_component_safe :: "(_) object_ptr set ⇒
  (_) object_ptr set ⇒ (_) heap ⇒ (_) heap ⇒ bool"
where
  "is_weakly_dom_component_safe S_arg S_result h h' = (
    let removed_pointers =
      fset (object_ptr_kinds h) − fset (object_ptr_kinds h') in
    let added_pointers =
      fset (object_ptr_kinds h') − fset (object_ptr_kinds h) in
    let arg_components =
      (⋃ptr ∈ (⋃ptr ∈ S_arg. set |h ⊢ get_dom_component ptr|_r) ∩
      fset (object_ptr_kinds h). set |h ⊢ get_dom_component ptr|_r) in
    let arg_components' =
      (⋃ptr ∈ (⋃ptr ∈ S_arg. set |h ⊢ get_dom_component ptr|_r) ∩
      fset (object_ptr_kinds h'). set |h' ⊢ get_dom_component ptr|_r) in
    removed_pointers ⊆ arg_components ∧
    S_result ⊆ arg_components' ∪ added_pointers  ∧
    (∀outside_ptr ∈ fset (object_ptr_kinds h) ∩ fset (object_ptr_kinds h')
      − (⋃ptr ∈ S_arg. set |h  ⊢ get_dom_component ptr|_r).
      preserved (get_M outside_ptr id) h h'))"
```

The only two differences are that we do not put any constraint upon the `added_pointers`, and that weaken the restriction on s_{result} by also allowing the method to return added pointer that are *not* part of the argument components. In other words, *weak DOM-Component safety* allows methods to additionally create and return new pointers, covering, for example, constructors.

6.3 Component Safety Classification of the DOM Methods

In the following, we will discuss to what extend 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, especially `append_child`, break our separation in unexpected ways. An overview of the safety levels of the DOM methods is shown in Table 6.1.

6.3.1 Getters and Simple Recursive DOM Methods

Most simple DOM methods are strongly DOM-Component safe. By simple, we mean methods that basically iterate the node tree along the `childNodes` relation. We have already seen that `get_child_nodes` is strongly DOM-Component safe, but omitted the proof idea. Essentially, the proof utilizes the fact that the definition of a DOM-Component is built upon `to_tree_order`, which simply iterates the tree by calling `get_child_nodes` recursively. Therefore, it is hardly surprising that for any object pointer, its children reside within the same component as the pointer itself, which follows from `get_child_nodes_strongly_dom_component_safe`. The method `get_parent` is also strongly DOM-Component safe, which follows from a similar argument.

The recursive variants, `get_root_node`, `get_element_by_id`, and similar methods, are also strongly DOM-Component-safe. For example, we state the lemma for the method `get_element_by_id` as follows:

```
lemma get_element_by_id_is_strongly_dom_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_dom_component_safe {ptr} {cast result} h h'"
```

Note that 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 group of lemmas is similar; 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_dom_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 DOM-Component safety.

Table 6.1: Classification of the most important 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	strong
`assigned_slot`	unsafe	strong
`get_host`	unsafe	unsafe
`get_composed_root_node`	unsafe	unsafe
`assigned_nodes`	unsafe	unsafe
`get_owner_document`	unsafe	unsafe
`adopt_node`	unsafe	unsafe
`remove_child`	unsafe	unsafe
`insert_before`	unsafe	unsafe
`append_child`	unsafe	unsafe

6.3.2 Constructors

In general, methods that create new objects are weakly DOM-component safe, because most of them return a new object which is not part of any component yet, since it has just been created. Our constructors 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).

First, we will show that they are *not* strongly DOM-Component safe. For example, for `create_element` we show the following lemma:

```
lemma create_element_not_strongly_dom_component_safe:
  obtains h :: heap_final 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_dom_component_safe
    {cast document_ptr} {cast new_element_ptr} h h'"
```

On a technical level, the structure of this kind of lemma is different from the lemmas showing the safety; we use the Isabelle **`obtains`** construction to show that there exists at least one heap for which the method is not (strongly or weakly) DOM-Component safe. It is also important to note that we prove this lemma outside of our locale construction (for more details, refer to Section 4.2)—we do so because we will use concrete heaps with closed extension slots (hence the usage of `heap_final`) in this kind of counter-example proof, on which we will symbolically evaluate a range of our DOM methods. Since inside our locales, we only have a few concrete definitions available, we conduct the proof outside, where the full range of exported definitions is available. The overhead of potentially duplicating this kind of proof over multiple extensions of our type universe is small, however, as we will see now in the case of the proof of `create_element_not_strongly_dom_component_safe`:

```
proof −
  let ?h0           = "Heap fmempty :: heap_final"
  let ?P            = "create_document"
  let ?h1           = "|?h0 ⊢ ?P|h"
  let ?document_ptr = "|?h0 ⊢ ?P|r"

  show thesis
    apply(rule that[where
      h               = "?h1" and
      document_ptr    = "?document_ptr" and
      new_element_ptr = "|?h1 ⊢ create_element ?document_ptr|r" and
      h'              = "|?h1 ⊢ create_element ?document_ptr|h"])
    by code_simp+
qed
```

The proof works as follows: First, we programmatically construct a heap that will serve as our counterexample, starting from the empty heap `Heap fmempty` (recall that our heap datatype is basically a finite map, hence the use of `fmempty`). On this empty heap, we then invoke `create_document` to create the main document. The resulting new heap and document pointer are then bound to the proof-local variables `h` and `document_ptr`, respectively. Before we can obtain them, however, Isabelle requires us to prove that the given program terminates without error and such heap and document pointer actually exist. Since we have a concrete start heap given and access to all method definitions, we can simply use `code_simp` to symbolically execute the statement and show that it evaluates to true (for more information about our use of the code generator, see Section 2.1.5). Finally, we can plug our concrete `h` and `document_ptr` into `that`, which refers to the lemma statement and has schematic placeholders for all arbitrary, but fixed variables. We do not need to fill in every variable, as the simplifier will be able to calculate them automatically, or ignore them if they are not necessary to proof the lemma. Then, we can invoke simplification by using code theorems (i. e., theorems that Isabelle generates from definitions to use them for code generation) again to finish the proof.

It is interesting to note that this proof pattern allows us to work with concrete heaps without ever having to state them manually, which would be tedious and clutter our proof text, especially for larger heaps. If we inspect the proof status of Isabelle, we will see the concrete values of both `h` and `document_ptr`:

```
document_ptr = document_ptr.Ref 1 ∧
h = Heap (fmap_of_list [
  (object_ptr.Ext (Inr (Inl (document_ptr.Ref 1))),
  (|RObject.nothing = (),
    ... = Inr (Inl (|RDocument.nothing = (), doctype = [],
    document_element = None, disconnected_nodes = [], ... = None|))|))
])
```

Nevertheless, for most of our non-safety proves we will use the formalisation of our running example in Figure 6.1 (created using a list of mappings from pointers to objects) as our base counter-example heap instead of the empty heap, which allows us to understand the counter-example more easily, especially when a more complex one is needed. In the end, any approach proves the same lemma in Isabelle, so it is more a matter of personal preference.

Even though `create_element` is not strongly DOM-Component safe, it is weakly safe:

```
lemma create_element_is_weakly_dom_component_safe:
  assumes "valid_heap h"
  assumes "h ⊢ create_element document_ptr tag →r result →h h'"
shows "is_weakly_dom_component_safe {cast document_ptr} {cast result} h h'"
```

The proof idea is that the only object (that existed in `h`) that gets changed is `document_ptr`, which adds the newly created element to its list of disconnected nodes. The new element pointer forms its own DOM-Component and therefore cannot interfere with any already existing ones.

The proofs and counter-proofs for the other constructors work in a similar manner and are all contained in the full Isabelle proof document.

6.3.3 Shadow Root Methods

The next class of methods includes ones that concern shadow roots or slotting directly, which are, in general, unsafe, which is expected. In case of a closed shadow tree (`mode` is set to `Closed`), methods trying to look inside Shadow Root DOM-Components (i. e., `get_shadow_root` and `assigned_slot`) will return an error, making them strongly DOM-component-safe in this case. Unaffected by the mode are methods trying to break out (i. e., `get_host`, `get_composed_root_node`, and `assigned_nodes`), thus they remain unsafe. For example, the lemma showing that `assigned_slot` is, in general, not weakly DOM-Component safe (and therefore also not strongly safe), can be stated as follows:

```
lemma assigned_slot_not_weakly_dom_component_safe:
  obtains h :: dom_final 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 for `create_element_not_strongly_dom_component_safe`, but now for the predicate `is_weakly_component_safe`. We use `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–whether a slot has been found or not. 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, with the difference that we now use our running example as the counter-example heap:

```
proof −
  let ?h = fancy_tabs_heap
  let ?node_ptr = "|?h ⊢ do {
    elements ← get_elements_by_tag_name fancy_tabs_document ''fancy−tabs'';
    children ← get_child_nodes (cast (elements ! 0));
    return (children ! 0)
  }|r"
  show thesis
    apply(rule that[where
      h        = "?h" and
      node_ptr = "?node_ptr" and
      slot_opt = "|?h ⊢ assigned_slot ?node_ptr|r" and
      h'       = "|?h ⊢ assigned_slot ?node_ptr|h"
    ])
    by code_simp+
qed
```

We bind our running example heap to `?h`, and then obtain the first child of the `<fancy-tabs>` element and bind it to `?node_ptr`. We know that `assigned_slot` will return one of the slots in T_2, which we do not need to state here explicitly as the code generator is able to calculate the correct slot itself. Since the argument for our method call, `?node_ptr`, came from T_1, we have shown that `assigned_slot` is indeed unsafe.

While `assigned_slot` is unsafe in general, we can show that for the special case that all shadow roots in the current heap are closed, `assigned_slot` is indeed strongly DOM-Component safe:

```
lemma assigned_slot_strongly_dom_component_safe:
  assumes "valid_heap h"
  assumes "h ⊢ assigned_slot node_ptr →r slot_opt →h h'"
  assumes "∀shadow_root_ptr ∈ fset (shadow_root_ptr_kinds h).
    h ⊢ get_mode shadow_root_ptr →r Closed"
shows "is_strongly_dom_component_safe
  {cast node_ptr} (cast ` set_option slot_opt) h h'"
```

The proof basically follows from the definition of `assigned_slot`, which checks whether the involved shadow root has the `Open` flag set. If not, the method returns `None`, which—together with the fact that `assigned_slot` does not modify the heap–makes it trivially strongly DOM-Component safe.

A similar argument can be made for `get_shadow_root`. The other methods of this category, `get_host`, `get_composed_root_node`, and `assigned_nodes` are always unsafe, which we show with a similar counter-example heap to the one used in the proof of `assigned_slot_not_weakly_dom_component_safe`, with the exception that the counter-example works even if the shadow root is closed.

6.3.4 Heap-Modifying and Global Methods

The last category of DOM methods includes the methods `adopt_node`, `remove_child`, `insert_before`, `append_child` and `get_owner_document`, which involve multiple pointers and traverse the node tree in various ways, and some also modify it in different locations. Surprisingly and unfortunately, they all operate across DOM-Component boundaries in unexpected ways, making them all *unsafe*. From a web application development perspective, 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. If a library developer were to use this method for setting up their component, they might inadvertently break out and change objects outside of their component.

The example heap that we use for the counter-example proof is the same for each DOM method of this category: In fact, we can use the heap of our running example (Figure 6.1) again to proof that the methods are unsafe:

- `get_owner_document`, called on any of the nodes inside the Shadow Tree (T_2) will return the root document, which lies outside the shadow root DOM-Component.

- `adopt_node` and `remove_child`, called on any of the nodes inside T_2, will remove the node from the component and add it to the list of the root document's disconnected nodes, thus modifying an object (the document) outside of T_2.
- `insert_before` and `append_child`, used to add the disconnected node in T_3 to any node inside the Shadow Tree T_2, will again modify the document in order to remove the node from its disconnected node list, which is outside of both T_2 and T_3.

The Isabelle proofs follow the same pattern as the other proofs showing the non-weak DOM-Component safety.

The fact that these important DOM methods are unsafe with regards to DOM-Components is undesirable, as this means that these methods break the expectations that a developer might have when working with shadow root components.

Summary

In this chapter, we have seen that 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 provide 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. In the next chapter, we will explore these and other possible remedies in more detail.

7 Beyond the Standard: Safe Web Components

If one examines the DOM methods from the previous section that we deem unsafe more closely, it becomes apparent that the owner document is of special importance to many operations. Therefore, if one could ensure that the owner document is always inside the same DOM-component as the other nodes involved, then we would not easily break out anymore.

In this chapter, we will explore ways how we could achieve that. We will see how small changes to our previous definition of components and the specification of the DOM itself lead to a situation where our DOM methods fulfil our expected (refined) notion of safety, meaning only those DOM methods violate the component boundary that cross the `shadowRoot` boundary explicitly. Therefore, the motivation underlying all changes proposed in this chapter is the search for a *better* component model that allows the safe composition of components within the DOM.

These changes, however small, are not backed by the informal specification document of the official DOM, even though, as we will later see, our changes still pass the compliance test suite. All considerations, definitions, and proofs in this section are therefore beyond what the official specification at the time of writing encompasses, but are to be understood as a proposal for enhancing the standard. All previous proofs and definitions, unless mentioned otherwise, remain valid. All Isabelle definitions and lemmas presented in this chapter can be found online; the modified Core DOM theories [17], the modified Shadow DOM theories [15], and the safely composable web component theories [10].

7.1 Making the DOM Safely Composable

Before we can refine our components and their definition of safety, we need to make some changes to the specification of the DOM itself. Intuitively, we will strengthen the separation between shadow root components and their surrounding sub-tree against unexpected traversal from within the shadow root component. In order to achieve this, we will "promote" our `ShadowRoot` class to a `Document`, which means we will change our type hierarchy such that our `ShadowRoot` inherits from `Document` instead of `Object`. As a result, `ShadowRoot` instances will then be able to function as owner documents for all their nodes.

It is interesting to note that with these proposed changes, and all other changes that become necessary, our changed specification remains compliant to the official DOM standard, as it still passes the official compliance test suite–however, we consider the reason for this to be an underspecified test suite.

In the following, we will first see which changes are necessary to our datatypes, followed by our heap invariants and general methods. Finally, we will summarise our changes to the DOM, the result of which we will call *Safely Composable Document Object Model (SCDOM)*.[1]

7.1.1 Data Model

The following definition shows our new `RShadowRoot`. Besides being compatible with all methods that require a document, the shadow root also gains the attributes `doctype`, `document_element`, and a `disconnected_nodes` list.

```
record ('node_ptr, 'element_ptr, 'character_data_ptr) RShadowRoot
  = "('node_ptr, 'element_ptr, 'character_data_ptr) RDocument" +
   nothing :: unit
   mode :: shadow_root_mode
   child_nodes :: "('node_ptr, 'element_ptr, 'character_data_ptr) node_ptr list"
```

Note that this change is an actual change of our theory files and *not* an extension as we have done when we introduced shadow roots originally into the *fDOM*. This is done because we do not want to keep any definitions or lemmas that were defined with a dependency on the old shadow root definition. Figure 7.1 shows our running example with the updated shadow root node, which is now much closer to being a document, i. e., it can have its own disconnected nodes.

7.1.2 Heap Invariants

All heap invariants from the DOM remain valid and necessary to describe a valid heap.

Only one additional requirement is needed: We need the relation induced by the `disconnected_nodes` attribute to remain acyclic, too. This has not been necessary so far because the only node that could have disconnected nodes was the root node, so acyclicity was guaranteed by the type system. We therefore need to introduce a new predicate that preserves this guarantee even when shadow roots, which are in the middle of the node tree, gain `disconnected_nodes`. We therefore define a new relation and augment our well-formedness predicate:

```
definition ptr_disconnected_node_rel :: "(_) heap
    ⇒ ((_) object_ptr × (_) object_ptr) set"
where
  "ptr_disconnected_node_rel h = (λ(x, y). (cast x, cast y))
     ` {(document_ptr, disconnected_node).
    document_ptr |∈| document_ptr_kinds h ∧
    disconnected_node ∈ set |h ⊢ get_disconnected_nodes document_ptr|ᵣ}"
```

[1] On a technical level, we keep all changes necessary for the SCDOM in a separate software repository, which is a fork of the one that contains the DOM. This allows us to easily compare both versions and to add changes to both variants in a similar way, where applicable, keeping the changes documented and small.

Finally, we can define our predicate to ensure that our instance of an SCDOM is wellformed:

```
definition heap_is_wellformed :: "(_) heap ⇒ bool"
where
  "heap_is_wellformed h ⟷ CoreDOM.heap_is_wellformed h ∧
    acyclic (parent_child_rel h ∪ host_shadow_root_rel h
      ∪ ptr_disconnected_node_rel h) ∧
    all_ptrs_in_heap h ∧
    distinct_lists h ∧
    shadow_root_valid h"
```

7.1.3 Methods

Before we can continue with describing the necessary changes to our DOM methods, we need to introduce a new helper construct, `get_ancestors_di`, which enumerates all ancestors of a given node in a *document-including* manner. We define it similarly to `get_ancestors` and `get_ancestors_si`:

```
partial_function (dom_prog) get_ancestors_di
    :: "(_::linorder) object_ptr ⇒ (_, (_) object_ptr list) dom_prog"
where
  "get_ancestors_di ptr = do {
    check_in_heap ptr;
    ancestors ← (case cast_object_ptr2node_ptr ptr of
      Some node_ptr ⇒ do {
        parent_ptr_opt ← get_parent node_ptr;
        (case parent_ptr_opt of
          Some parent_ptr ⇒ get_ancestors_di parent_ptr
        | None ⇒ do {
            document_ptr ← get_disconnected_document node_ptr;
            get_ancestors_di (cast document_ptr)
          })
      }
    | None ⇒ (case cast ptr of
        Some shadow_root_ptr ⇒ do {
          host ← get_host shadow_root_ptr;
          get_ancestors_di (cast host)
        } |
        None ⇒ return []));
    return (ptr # ancestors)
  }"
```

For the most part, we can use the definitions of our DOM methods and their proofs also in the SCDOM. However, we will need to make small changes to three of them, so

that they preserve our new wellformed-ness predicate. In the following, we will describe these necessary changes.

Before we can safely remove a shadow root from our heap, we now need to additionally ensure that it does not have any disconnected nodes left. Otherwise, we would violate the heap well-formedness condition that all nodes either have a parent or are a disconnected node recorded in a document. We have highlighted the changes compared to the version in the DOM:

```
definition remove_shadow_root :: "(_) element_ptr ⇒ (_, unit) dom_prog"
 where
  "remove_shadow_root element_ptr = do {
    shadow_root_ptr_opt ← get_shadow_root element_ptr;
    (case shadow_root_ptr_opt of
      Some shadow_root_ptr ⇒ do {
        children ← get_child_nodes (cast shadow_root_ptr);
        disconnected_nodes ← get_disconnected_nodes
          (cast shadow_root_ptr);
        (if children = [] ∧ disconnected_nodes = []
        then do {
          set_shadow_root element_ptr None;
          delete_M shadow_root_ptr
        } else do {
          error HierarchyRequestError })
      } |
      None ⇒ error HierarchyRequestError)}"
```

Another method that requires changes is `adopt_node`. Previously, we were able to re-use `adopt_node` from our DOM. Now, however, with the addition of a document that is not guaranteed to be a root node any longer, `adopt_node` is able to introduce cycles into our node tree. Therefore, we add a check that fails if the node that is to be adopted is already in the document's document-including ancestors. This check is very similar to the one used in `insert_before`. Our new definition looks as follows:

```
definition adopt_node :: "(_) document_ptr ⇒ (_) node_ptr ⇒ (_, unit) dom_prog"
where
  "adopt_node document node = do {
    ancestors ← get_ancestors_di (cast document);
    (if cast node ∈ set ancestors
      then error HierarchyRequestError
      else CoreDOM.adopt_node document node)}"
```

The definition of `get_owner_document` for shadow roots also needs to change. Since a shadow root inherits from a document now, we want it to behave like a document, which means it is no longer appropriate to continue searching the tree upwards for the next document. Therefore, we change it to simply to return itself, which is what we do for documents as well:

```
definition get_owner_document_shadow_root_ptr :: "(_) shadow_root_ptr ⇒ unit
  ⇒ (_, (_) document_ptr) dom_prog"
where
  "get_owner_document_shadow_root_ptr shadow_root_ptr
    = CoreDOM.get_owner_document_document_ptr (cast shadow_root_ptr)"
```

7.2 A New Kind of Component

Our previous definition of components and their safety carries over to our SCDOM without issue, meaning that the list of nodes for every given component does not change based on whether we use the DOM or SCDOM. However, the switch to the SCDOM alone makes our methods only a little safer; while, for example, `append_child`, called from within a shadow root component, will now completely stay inside that same component since its shadow root is now also its owner document, we gain nothing if one of the arguments is inside a disconnected component, which will never contain its owner document.

This observation leads us to the second necessary change: One possible way to achieve strong component safety for the DOM methods in question is to define a new kind of component that includes all nodes which are reachable in tree order from the *owner document* instead of the root node as we did for DOM-components. On first glance, it might seem that we simply broaden our definition of safety. However, if we accept that a disconnected component shares the same level of trust as the component of its owner document, then our modified definition of a component captures our intention more precisely. Alas, we define components in our new model as follows:

Definition 7.2.1 (SCDOM-Component)**.** *An* SCDOM-Component *of pointer p is the list of all pointers reachable in shadow-excluding tree order starting from the owner document of p, plus all pointers that are reachable in shadow-excluding tree order starting from each disconnected node of p. Formally, we define it as follows:*

```
definition get_scdom_component:: "(_) object_ptr
  ⇒ (_, (_) object_ptr list) dom_prog"
where
  "get_scdom_component ptr = do {
    document ← get_owner_document ptr;
    disc_nodes ← get_disconnected_nodes document;
    tree_order ← to_tree_order (cast document);
    disconnected_tree_orders ← map_M (to_tree_order ∘ cast) disc_nodes;
    return (tree_order @ (concat disconnected_tree_orders))
  }"
```

Note that one could also attempt to apply the definition of SCDOM-Components to the standard DOM; however, since the shadow roots there do not have their own owner document, our component definition would collapse into one big component per

document, which would include all containing shadow roots and their sub-trees, which is too much.

The idea is very similar to the one of our normal components; instead of just traversing the `childNode`-relation, we also traverse the `disconnectedNode`-relation. Therefore, `get_owner_document` brings us to the root of that sub-tree, from which on we include all node in tree-order reachable from the document itself, or from any of its disconnected nodes.

This component definition distinguishes only two different types of components: document and shadow root components. Compared to our DOM-components, we do not have a disconnected component anymore, since our definition is broader and does not stop traversing the tree upwards at a `Node` anymore. Alas, we define:

Definition 7.2.2 (Document SCDOM-Component)**.** *A* Document SCDOM-Component *is an SCDOM-component whose root node is of type* `Document`*, but* not *of type* `ShadowRoot`*. Formally, we define:*

```
definition is_document_scdom_component :: "(_) object_ptr list ⇒ bool"
where
  "is_document_scdom_component c =
    is_document_ptr_kind (hd c) ∧ (¬is_shadow_root_ptr_kind (hd c))"
```

Again, this kind of component captures the "main part" of the tree, as this kind of node still can only occur at the very top.

Definition 7.2.3 (Shadow Root SCDOM-Component)**.** *A* Shadow Root SCDOM-Component *is an SCDOM-component whose root node is of type* `ShadowRoot`*. Formally, we define:*

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

This definition is in line with the one for the DOM, and again can be considered the "canonical" component.

We define *SCDOM-Component safety* in a similar way to *DOM-Component safety*, but use the SCDOM-component instead of the DOM-component.

Definition 7.2.4 (SCDOM-Component Safety)**.** *A DOM method is* SCDOM-Component safe *if and only if all returned pointers and modifications are confined to exactly those SCDOM-Components given by its arguments. In Isabelle, we define:*

```
definition is_strongly_scdom_component_safe :: "(_) object_ptr set ⇒
  (_) object_ptr set ⇒ (_) heap ⇒ (_) heap ⇒ bool"
where
  "is_strongly_scdom_component_safe S_arg S_result h h' = (
    let removed_pointers =
      fset (object_ptr_kinds h) − fset (object_ptr_kinds h') in
```

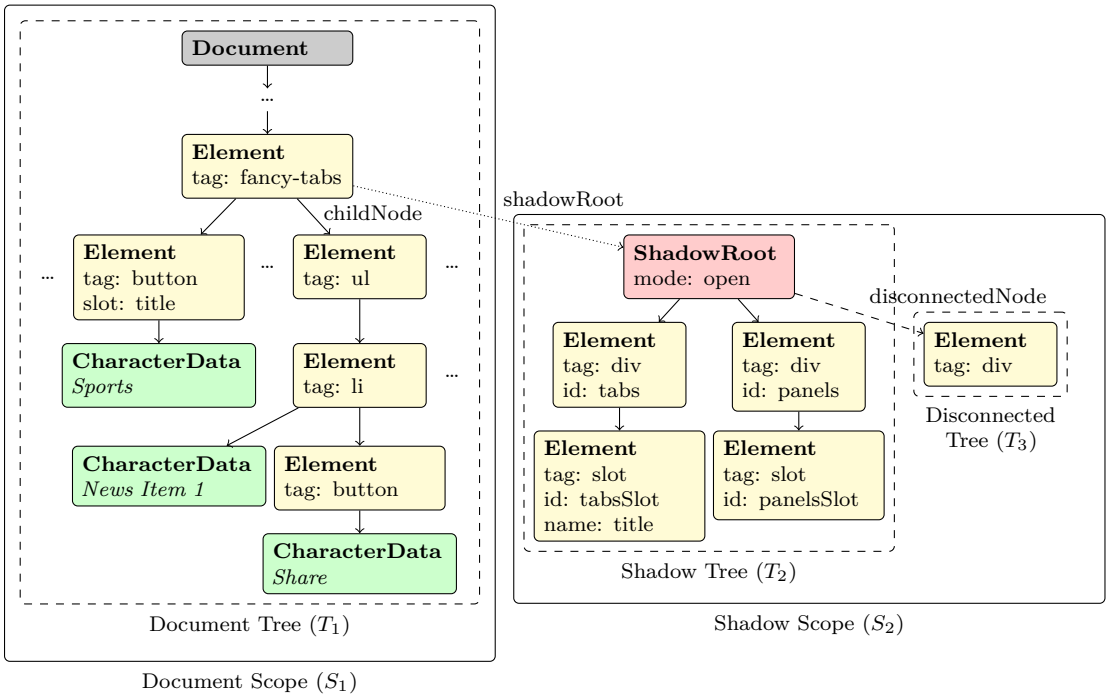


Figure 7.1: SCDOM-Components also include any potential disconnected tree. Here, S_2 includes both T_2 and T_3.

```
let added_pointers =
  fset (object_ptr_kinds h') − fset (object_ptr_kinds h) in
let arg_components =
  (⋃ptr ∈ (⋃ptr ∈ S_arg. set |h ⊢ get_scdom_component ptr|_r) ∩
  fset (object_ptr_kinds h). set |h ⊢ get_scdom_component ptr|_r) in
let arg_components' =
  (⋃ptr ∈ (⋃ptr ∈ S_arg. set |h ⊢ get_scdom_component ptr|_r) ∩
  fset (object_ptr_kinds h'). set |h' ⊢ get_scdom_component ptr|_r) in
removed_pointers ⊆ arg_components ∧
added_pointers ⊆ arg_components' ∧
S_result ⊆ arg_components' ∧
(∀outside_ptr ∈ fset (object_ptr_kinds h) ∩ fset (object_ptr_kinds h') −
  (⋃ptr ∈ S_arg. set |h  ⊢ get_scdom_component ptr|_r).
  preserved (get_M outside_ptr id) h h'))"
```

The formal definition is similar to the one given for strong DOM-component safety, with the difference that we use `get_scdom_component` instead of `get_dom_component`.

For SCDOM-Components, we do not need to treat DOM methods that create new object separately, as we will see that we are able to proof that they are SCDOM-Component safe.

Figure 7.1 shows the difference between DOM-Components and SCDOM-Components using our running example in the SCDOM model. Since the main difference between both kinds of components is that any potential disconnected tree is now included, the document DOM-Component T_1 is equivalent to its surrounding document SCDOM-component S_1, since their root node was already identical to their owner document. T_2 and T_3, however, are now both captured by a single shadow root SCDOM-component $S_2 = T_2 \cup T_3$.

7.3 General Properties of SCDOM-Components

Now that we have ensured that our new components are well defined, we will have a closer look at some important characterisations, especially regarding the relationship between DOM-Components and SCDOM-Components. Figure 7.1 gave us already an intuition, but we want to be more precise.

First of all, we can show that for all nodes, the nodes of their DOM-Component form a subset of those from their SCDOM-Component. In Isabelle, we prove the following lemma:

```
lemma get_scdom_component_subset_get_dom_component:
  assumes "valid_heap h"
  assumes "h ⊢ get_scdom_component ptr →r sc"
  assumes "h ⊢ get_dom_component ptr →r c"
shows "set c ⊆ set sc"
```

For any given node, its root node either inherits from `Document`, in which case it is both the node's owner document and root node, or it is not, in which case it must inherit from `Node` and be disconnected. In the first case, SCDOM-Component and DOM-Component are identical; in the second case, the root must be in the list of disconnected nodes from its owner document, which is covered by our definition of a SCDOM-Component.

Furthermore, we can show that all nodes are within the same SCDOM-Component as their owner document:

```
lemma get_scdom_component_owner_document_same:
  assumes "valid_heap h"
  assumes "h ⊢ get_scdom_component ptr →r sc"
  assumes "ptr' ∈ set sc"
  obtains owner_document where
    "h ⊢ get_owner_document ptr' →r owner_document" and
    "cast owner_document ∈ set sc"
```

This is another important property, without which we would not be able to justify the term "component". The main proof observation is that all children have the same owner document as their parents, and therefore all pointers in the list returned by `to_tree_order` share the same owner document.

Another interesting property is that there can be at most one owner document per SCDOM-Component:

```
lemma get_scdom_component_different_owner_documents:
  assumes "valid_heap h"
  assumes "h ⊢ get_owner_document ptr →r owner_document"
  assumes "h ⊢ get_owner_document ptr' →r owner_document'"
  assumes "owner_document ≠ owner_document'"
shows "set |h ⊢ get_scdom_component ptr|r ∩
    set |h ⊢ get_scdom_component ptr'|r = {}"
```

We state this property as follows: given two `get_owner_document` calls of pointers `ptr` and `ptr'`, that return different owner documents, we show that the SCDOM-Components of `ptr` and `ptr'` are completely disjunct. The proof mainly uses the observation that all pointers in such a component must have the same owner document.

7.4 Updated DOM Method Classification

With the new definition of SCDOM-Components, we can now re-classify our DOM methods accordingly. The overview of results are shown in Table 7.1.

Most notably, `adopt_node`, `remove_child`, `insert_before`, and `append_child` are now strongly SCDOM-Component-safe, which therefore fulfills the requirements with which we started our improvement efforts.

For example, the lemma for `adopt_node` looks as follows:

```
lemma adopt_node_is_strongly_scdom_component_safe:
  assumes "valid_heap h"
  assumes "h ⊢ adopt_node document_ptr node_ptr →r node_ptr2 →h h'"
shows "is_strongly_scdom_component_safe
    {cast document_ptr, cast node_ptr} {cast node_ptr2} h h'"
```

The proof idea uses the fact that the disconnected node of `node_ptr` is now always within the same SCDOM-Component as `node_ptr` itself.

The situation for our constructor methods has also improved; `create_element` and `create_character_data` are now strongly SCDOM-Component safe, which is desirable. The other two constructors, `attach_shadow_root` and `create_document` are still only weakly SCDOM-Component safe, however, as they both still create new SCDOM-Components. Therefore, we could not really expect an improvement there.

Summary

In this chapter, we set out to find a solution to the unsatisfactory guarantees provided by standard shadow roots. The idea to promote them to a full-fledged document is not revolutionary–in the official standard, they are already a `DocumentFragment`, which indicates already some level of independence. However, using our formalisation, we could explore this change using a simplified model on a formal basis. We learned that a few changes to the model are necessary to preserve the guarantees that we carefully build in the previous chapter, but overall, these changes are small. In the end, we achieved our

Table 7.1: Classification of the most important DOM methods into whether they are SCDOM-component safe or not. 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`	strong	strong
`create_character_data`	strong	strong
`create_document`	weak	weak
`attach_shadow_root`	weak	weak
`get_shadow_root`	unsafe	strong
`assigned_slot`	unsafe	strong
`get_host`	unsafe	unsafe
`get_composed_root_node`	unsafe	unsafe
`assigned_nodes`	unsafe	unsafe
`get_owner_document`	strong	strong
`adopt_node`	strong	strong
`remove_child`	strong	strong
`insert_before`	strong	strong
`append_child`	strong	strong

goal of confining all non-shadow root related methods to our notion of component, even though we had to widen our component definition a bit.

Of course, the official standard associates a lot more with a document than we do, such as an origin, i. e., the URL of the website. Would a shadow root then also get all these attributes, and if yes, how would they be set? For example, one could think about whether the origin of a shadow root component could even be related to the author of component. We will leave these considerations for future work.

8 Related Work

The work presented in this thesis combines approaches from multiple disciplines; we combine interactive theorem proving in a higher-order logic with real-world software standards, test suites, and component-based software engineering. We therefore group the related work into multiple, loosely related sections.

8.1 Formal Models of Software Standards

Most software in existence has been specified prior to implementation in one way or another, ranging from a few use cases that have been sketched out in natural English for a small utility script to full-fledged requirements engineering and a specification of all APIs that has been machine-checked for inconsistencies. The first part of this thesis seeks to improve upon one specific kind of specification for one specific kind of software: The Document Object Model, which is an *interface* that can be used to work with tree-based structures and which is specified by a description of its algorithms and APIs in *structured English* contained in a *standardization document*. Our approach is therefore best compared with efforts that share these three criteria. We consider therefore, for example, specifications of programming languages such as ECMAScript as *out-of-scope*, as well as means of specifying concrete software products such as the Java Modeling Language (JML) [45] or the Z notation [1], since we believe these approaches are too different to be successfully compared to ours.

The most relevant works are ones that also address the DOM standard. There is a number of authors who also identified the DOM as one of the center pieces of web browser security, and thus chose to focus their formalisation efforts on it. Gardner et al. [30, 29, 62, 59] propose a non-executable, non-extensible, and non-mechanized operational semantics of a minimal DOM (Core DOM Level 1) and show how this semantics can be used for Hoare-style reasoning for analysing heaps of DOMs. The authors focus on providing a formal foundation for reasoning over client-side JavaScript programs that modify the DOM; JavaScript is out of our scope, and unfortunately their logic is not powerful enough for our considerations around node trees and their shapes and conditions. In addition, neither of these works defines formally the concept of web components nor the definition and verification of component safety properties.

In a similar spirit, the authors of Featherweight Firefox [7] create a simpler operational specification of the node-tree related DOM API methods. They attempt to provide a more complete formal model of a browser by including "cookies, HTTP requests and responses, user input, and a minimal scripting language" in their model, with similar characteristics as the one of Gardner et al..

Other DOM formalisations have been proposed in [33, 39], which are, however, rather lightweight and imprecise. For example, they can not distinguish multiple elements in a DOM instance with the same ID. This is due to the authors focusing on providing static analysis for JavaScript web applications, whereas we focus on the DOM side.

Another formal model of the DOM can be found in [57], which has also been created to support static analysis of JavaScript programs. The authors' framework allows for exchanging their DOM model with other ones, such as the one from GATEKEEPER [33] and TAJS [39]. Still, their DOM model is tightly coupled with their static analysis framework. Another contribution of [57] is an empirical study showing the most frequently used DOM fields and API methods on the top 10 000 websites, justifying our focus on those DOM API methods that we chose to model.

A different kind of DOM model is proposed by Lerner et al. [48]. The authors do not focus on JavaScript, but rather on the reactive nature of the DOM and therefore mainly model closely after the Events API of the DOM, which we exclude. Being a Redex [27] model, it is also executable, and the authors generated test cases from their model to find bugs related to events in real browsers. The authors also formulate invariants that a DOM instance will obey; however, they are mostly related to events and rather informal.

There are also other, related software specifications that we consider relevant. In [63], 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 formalisation of XML or XML-like data structures in Isabelle/HOL.

Another software standard that has been formalised in Isabelle/HOL is the Object Constraint Language (OCL) [18], which is a formalism that can be used to specify additional constraints on objects of UML diagrams. The formalisation has been used to find a number of inconsistencies in the OCL standard and is similar in many ways to *fDOM*. In particular, it uses a similar way of extending its type universe in an iterative manner without having to re-prove already proven properties.

8.2 Program Verification

Our approach shares many similarities with established approaches and logics from the area of program verification, especially separation logic [54]. Separation logic is an extension of Hoare logic [36] that mainly considers programs with lookup, update, and other operations on a heap memory model, e. g., programs using linked lists. It offers an assertion language with Hoare-style pre- and postconditions along with special rules that enables reasoning without having to consider the whole heap ("frame rule"). Our `reads` and `writes` predicates from Section 3.4.1 also capture precisely those pointers and even attributes that are read or modified by a particular DOM method, from simple getters to complex modifications in multiple locations. Together with our `reads_writes` lemmas we can then easily show that, for example, a `get_child_nodes` call will never be affected by `set_tag_name`, as they access different attributes of our data model. The focus of our work lies more on exploring questions such as: What is a useful measure of "local modifications" in the context of DOM programs, which needs to be answered

before we could even consider a more efficient calculus. Therefore, we consider our work complementary to separation logic and its advancements.

There have also been a number of efforts to introduce the benefits of annotating code with pre- and postconditions to industrial-scale programming systems. For example, Spec# [5], a programming system based on C#, uses the Boogie verifier [47] to permit the "specification and reasoning about object invariants even in the presence of callbacks". Such approaches would certainly be useful for giving developers of DOM programs better tools to reason about them, especially when it comes to pre- and postconditions that are concerned with the safety of DOM components as proposed by us. However, creating such industrial-scale reasoning tools that make use of our findings are certainly out of our scope currently.

8.3 Formal Approaches to Object-Orientation

The specification of the DOM makes heavy use of object-orientation to define the node-tree, the central data structure of the DOM, which can pose significant challenges for some formal methods. Our work shares a common goal with ownership type systems [24]. For example, [58] 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.

While the core idea of formalising object-oriented data models in an extensible way follows the construction presented in [19, 8], we differ significantly in aspects such as the modeling of typed pointers (references) and late binding of method invocations. In particular, we use a data model encoding based on Isabelle records, which are part of the standard framework and therefore well supported and documented, instead of creating a new datatype package which increases complexity. Also, we encoded DOM methods in a way that easily allows extensions by providing extension slots and modular proofs that can be reused, instead of a Java/C++ style method invocation table.

8.4 Iframes and Shadow Roots

Overall, shadow roots seem to achieve a very similar goal as the `<iframe>`-tag of the HTML standard. However, 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. Still, there have been attempts such as [66] to add policies to inter-iframe communication to enforce a client-side policy for iframe-based web mashups. We chose to approach the issue of web components in the DOM from the direction of shadow roots, as we believe their goal to be more closely aligned to what we consider a useful notion of components.

In [35], the authors present ShadowCrypt, a design paradigm for web developers to utilise shadow roots to secure and isolate user input fields on web sites. The attempt to achieve this mainly by altering some JavaScript properties on DOM objects and setting them to `null`. However, as shown in [28], there are many ways how the guarantees of ShadowCrypt can be broken; for example, by using less-frequently used input fields which are not covered by the tool. Others, such as [61], have also tried to use shadow roots for hiding additional information from client-side applications, but struggled in a similar way to [35].

Beeswax [46] is a similar approach to provide private components in real-world web browsers, but also relies on tediously keeping track on all different ways in which one can break the security assumptions of current shadow roots. As seen with ShadowCrypt, this is an error-prone process and requires constant updating in order to stay up-to-date with changing browser APIs.

[55] offers a more applied approach combining shadow roots to automatically turn multi-page applications into single-page applications.

These lines of work show that there is clearly a demand for secure web components that are less restrictive than iframes, but still offer certain security guarantees. We believe that current shadow roots are on the right track, but have ultimately a different goal [37], so we believe that more fundamental work, such as our SCDOM-Components, is necessary.

8.5 Components in General

Components are, of course, one of the pillar of any kind of software engineering. There are a lot of works, such as [41, 25], that formalise component-based *designs* and use a π-calculus and temporal logic together with model checking. This kind of approach as also been applied to components in the web ecosystem, for example by [56], where the author proposes a π-calculus-based model with operational semantics for more abstract web components, consisting of protocol specifications using interfaces such as ports. They work on a much higher level of abstraction and do not model the DOM, node tree, or tree manipulation language.

Other authors, such as those of [50, 43, 2], provide a more practical approach to web components by developing domain-specific languages or frameworks to make the use of shadow roots and related technologies easier, but also lack formal foundations.

In general, we use a very narrow definition of components, one that refers to sub-trees of nodes together with special nodes that guide the behavior of the involved functions. We believe that for this reasons, many other approaches to other definitions of components are therefore not applicable.

9 Conclusion and Future Work

In this thesis, we have seen multiple contributions that all achieved their common goal of providing a safe component system for the web. We have achieved this by using our formal model and analysis of the DOM with shadow roots to build a novel definition of web components. Using this definition together with our formal framework, we have provided proofs of important component separation properties to give recommendations for improvements of the standard and possible tools that aid the safe development of libraries. In addition, we have shown that these improvements would actually result in only small changes to the DOM standard, i. e., a small change in the type hierarchy of a shadow root together with changes in affected method definitions.

In more detail, we have extracted novel definitions of web components and their safety and have classified the most important DOM API methods accordingly. By doing so, we have uncovered that in particular methods involving the *owner document* are problematic and easily break out of what is supposed to be a component. We were able to conduct this kind of analysis by using our formal model of the DOM enriched with shadow roots, which serves as a robust and easily extendable formal framework and which we also used for other purposes, e. g., for extracting novel properties and invariants such as our well-formedness invariants that make the assumptions about the DOM much more explicit. With all these properties, we were then able to show that the proposed changes to the DOM are very small and manageable, improve the safety of the DOM API, and lead to a less ambiguous API. In order to ensure that all comments and suggestions regarding the official DOM standard are applicable, we have shown that our model complies to the official DOM standard. We have done so in the same way as, for example, implementations of the DOM in web browsers do, by symbolically evaluating all relevant test cases from the compliance test suite on our model. Finally, our formal analysis has been conducted inside the Isabelle/HOL framework, which is based on mathematical logic and which allowed us to build the formal model in an executable, extendable, re-usable and automatically checked way.

Concretely, besides clarifications regarding shadow roots and related algorithms such as slotting, we contribute a new kind of component definition with strong safety guarantees to the DOM standard: If shadow roots were to get traits from documents (such as disconnected nodes), they would provide components that are safe with regards to accidental access and modification from the most common DOM methods.

We see several lines of future work. We consider tightening the link between the formal specification and the actual implementations used by various web browsers as the most important line of future work. One promising approach to achieve this goal is the systematic generation of test cases from the formal specification using test case generation techniques that are integrated into Isabelle/HOL [20]. The generated test

cases can, as the already existing manually developed test cases, be used for validating the compliance of actual browser implementation and be added to the official compliance test suite.

Furthermore, there is another promising area with regards to extending the scope of our formalisation: using the extensibility of our formalisation to add support for `HTMLElement` (and its sub-types such as `HTMLIFrameElement`). As the concept of iframes is fundamental for restricting information flow between parts of a website originating from different security domains, such a formalisation would allow us to reason over web security properties in Isabelle/HOL and compare them with shadow roots. On the first glance, the two concepts do not have much in common. However, having a closer look reveals that the concepts are in fact closely related. Therefore, one could approach the goal of safe web components from both directions, meaning that, 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 formalise the core of the HTML standard on top of our DOM formalisation. This allows us to compare both concepts formally and also investigate the impact of adding security features to shadow roots.

Bibliography

[1] Jean-Raymond Abrial, Stephen A. Schuman and Bertrand Meyer. 'Specification Language'. In: *On the Construction of Programs*. 1980, pp. 343–410.

[2] Devdatta Akhawe et al. 'Data-Confined HTML5 Applications'. In: *Computer Security – ESORICS 2013*. Ed. by Jason Crampton, Sushil Jajodia and Keith Mayes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 736–754. ISBN: 978-3-642-40203-6.

[3] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*. Computer science and applied mathematics. Academic Press, 1986.

[4] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation*. URL: `https://web.archive.org/web/20191105181806/https://isabelle.in.tum.de/dist/Isabelle2019/doc/locales.pdf` (visited on 05/11/2019).

[5] Mike Barnett, Rustan Leino and Wolfram Schulte. 'The Spec# Programming System: An Overview'. In: *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*. Vol. 3362. Lecture Notes in Computer Science. Springer, Jan. 2005, pp. 49–69. URL: `https://www.microsoft.com/en-us/research/publication/the-spec-programming-system-an-overview/`.

[6] Eric Bidelman. *Shadow DOM v1: Self-Contained Web Components*. URL: `https://web.archive.org/web/20191014084634/https://developers.google.com/web/fundamentals/web-components/shadowdom` (visited on 14/10/2019).

[7] Aaron Bohannon and Benjamin C. Pierce. 'Featherweight Firefox: Formalizing the Core of a Web Browser'. In: *Proceedings of the 2010 USENIX Conference on Web Application Development*. WebApps10. USENIX Association, 2010.

[8] Achim D. Brucker. 'An Interactive Proof Environment for Object-oriented Specifications'. PhD thesis. ETH Zurich, Mar. 2007.

[9] Achim D. Brucker and Michael Herzberg. 'A Formal Semantics of the Core DOM in Isabelle/HOL'. In: *Companion Proceedings of the The Web Conference 2018*. WWW 18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 741–749. ISBN: 978-1-4503-5640-4. DOI: `10.1145/3184558.3185980`.

[10] Achim D. Brucker and Michael Herzberg. 'A Formalization of Safely Composable Web Components'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x.

[11] Achim D. Brucker and Michael Herzberg. 'A Formalization of Web Components'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x.

[12] Achim D. Brucker and Michael Herzberg. 'A Formally Verified Model of Web Components'. In: *Formal Aspects of Component Software - 16th International Conference on Formal Aspects of Component Software, 23-25 October 2019, Amsterdam, Proceedings*. 2019.

[13] Achim D. Brucker and Michael Herzberg. 'Formalizing (Web) Standards - An Application of Test and Proof'. In: *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*. Ed. by Catherine Dubois and Burkhart Wolff. Vol. 10889. Lecture Notes in Computer Science. Springer, 2018, pp. 159–166. DOI: 10.1007/978-3-319-92994-1_9.

[14] Achim D. Brucker and Michael Herzberg. 'Shadow DOM: A Formal Model of the Document Object Model with Shadow Roots'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x.

[15] Achim D. Brucker and Michael Herzberg. 'Shadow SC DOM: A Formal Model of the Safelty Composable Document Object Model with Shadow Roots'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x.

[16] Achim D. Brucker and Michael Herzberg. 'The Core DOM'. In: *Archive of Formal Proofs* (2018). https://www.isa-afp.org/entries/Core_DOM.html, Formal proof development. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x.

[17] Achim D. Brucker and Michael Herzberg. 'The Safely Composable DOM'. In: *Archive of Formal Proofs* (28th Sept. 2020). Formal proof development. Submitted. A snapshot taken at thesis submission time can be found under [21]. ISSN: 2150-914x.

[18] Achim D. Brucker and Burkhart Wolff. 'A Proposal for a Formal OCL Semantics in Isabelle/HOL'. In: *Theorem Proving in Higher Order Logics (TPHOLs)*. Lecture Notes in Computer Science 2410. Hampton, VA, USA: Springer-Verlag, 2002, pp. 99–114. ISBN: 3-540-44039-9. DOI: 10.1007/3-540-45685-6_8.

[19] Achim D. Brucker and Burkhart Wolff. 'An Extensible Encoding of Object-oriented Data Models in HOL'. In: *Journal of Automated Reasoning* 41.3 (Nov. 2008), pp. 219–249. ISSN: 1573-0670. DOI: 10.1007/s10817-008-9108-3.

[20] Achim D. Brucker and Burkhart Wolff. 'On theorem prover-based testing'. In: *Formal Aspects of Computing* 25.5 (Sept. 2013), pp. 683–721. ISSN: 1433-299X. DOI: 10.1007/s00165-012-0222-y.

[21] Achim Brucker and Michael Herzberg. *Formalized Web Components*. Nov. 2020. DOI: 10.5281/zenodo.4250021. URL: https://doi.org/10.5281/zenodo.4250021.

[22] Mike Champion et al. *Document Object Model (Core) Level 1*. 1998. URL: https://web.archive.org/web/20190928181328/https://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html (visited on 28/09/2019).

[23] Alonzo Church. 'A Formulation of the Simple Theory of Types'. In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. ISSN: 00224812. URL: http://www.jstor.org/stable/2266170.

[24] Dave Clarke et al. 'Ownership Types: A Survey'. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by Dave Clarke, James Noble and Tobias Wrigstad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 15–58. ISBN: 978-3-642-36946-9. DOI: 10.1007/978-3-642-36946-9_3.

[25] Antonio Coronato and Giuseppe DE Pietro. 'Formal Specification of Wireless and Pervasive Healthcare Applications'. In: *ACM Trans. Embed. Comput. Syst.* 10.1 (Aug. 2010), 12:1–12:18. ISSN: 1539-9087. DOI: 10.1145/1814539.1814551.

[26] Manuel Eberl et al. *Archive of Formal Proofs*. URL: https://www.isa-afp.org/ (visited on 05/11/2019).

[27] Matthias Felleisen, Robert Bruce Findler and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009.

[28] Michael Freyberger et al. 'Cracking ShadowCrypt: Exploring the Limitations of Secure I/O Systems in Internet Browsers'. In: *Proceedings on Privacy Enhancing Technologies* 2018.2 (Apr. 2018), pp. 47–63. DOI: 10.1515/popets-2018-0012.

[29] Philippa A. Gardner et al. 'Local Hoare Reasoning About DOM'. In: *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS 08. Vancouver, Canada: ACM, 2008, pp. 261–270. ISBN: 978-1-60558-152-1. DOI: 10.1145/1376916.1376953.

[30] Philippa Gardner et al. 'DOM: Towards a Formal Specification'. In: *Proceedings of the ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X08)*. 2008.

[31] Google. *Google Chrome - The New Chrome & Most Secure Web Browser*. 2nd Sept. 2008. URL: https://web.archive.org/web/20191003155535/https://web-platform-tests.org/ (visited on 14/11/2019).

[32] Mike J. C. Gordon and Tom F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press, 1993. ISBN: 0-521-44189-7.

[33] Salvatore Guarnieri and Benjamin Livshits. 'GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code'. In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM09. Montreal, Canada: USENIX Association, 2009, pp. 151–168.

[34] Florian Haftmann and Lukas Bulwahn. *Code generation from Isabelle/HOL theories*. 9th June 2019. URL: https://web.archive.org/web/20191105181007/https://isabelle.in.tum.de/doc/codegen.pdf (visited on 05/11/2019).

[35] Warren He et al. 'ShadowCrypt: Encrypted Web Applications for Everyone'. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS 14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1028–1039. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660326.

[36] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[37] Hayato Ito. *Comment on bug report "Add "closed" flag to createShadowRoot (bugzilla: 20144)"*. 4th Dec. 2015. URL: https://web.archive.org/web/20190926124517/https://github.com/w3c/webcomponents/issues/100#issuecomment-161867941 (visited on 26/09/2019).

[38] Artur Janc and Lukasz Olejnik. 'Web Browser History Detection as a Real-World Privacy Threat'. In: *Computer Security – ESORICS 2010*. Ed. by Dimitris Gritzalis, Bart Preneel and Marianthi Theoharidou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 215–231. ISBN: 978-3-642-15497-3.

[39] Simon Holm Jensen, Magnus Madsen and Anders Møller. 'Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications'. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE 11. Szeged, Hungary: ACM, 2011, pp. 59–69. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025125.

[40] Sudeep Kanav, Peter Lammich and Andrei Popescu. 'A Conference Management System with Verified Document Confidentiality'. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 167–183. ISBN: 978-3-319-08867-9.

[41] Daniel Karlsson, Petru Eles and Zebo Peng. 'Formal verification of component-based designs'. In: *Design Automation for Embedded Systems* 11.1 (Mar. 2007), pp. 49–90. ISSN: 1572-8080. DOI: 10.1007/s10617-006-9723-3.

[42] Gerwin Klein et al. 'seL4: formal verification of an OS kernel'. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[43] Michael Krug and Martin Gaedke. 'SmartComposition: Bringing Component-based Software Engineering to the Web'. In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*. iiWAS 15. Brussels, Belgium: ACM, 2015, 63:1–63:4. ISBN: 978-1-4503-3491-4. DOI: 10.1145/2837185.2837247.

[44] Yves Lafon and Travis Leithead. *WebIDL Level 1*. 15th Dec. 2016. URL: https://web.archive.org/web/20190924190043/https://www.w3.org/TR/WebIDL-1/ (visited on 24/09/2019).

[45] Gary T. Leavens, Albert L. Baker and Clyde Ruby. 'JML: A Notation for Detailed Design'. In: *Behavioral Specifications of Businesses and Systems*. Ed. by Haim Kilov, Bernhard Rumpe and Ian Simmonds. Boston, MA: Springer US, 1999, pp. 175–188. ISBN: 978-1-4615-5229-1. DOI: 10.1007/978-1-4615-5229-1_12. URL: https://doi.org/10.1007/978-1-4615-5229-1_12.

[46] Jean-Sébastien Légaré, Robert Sumi and William Aiello. 'Beeswax: a platform for private web apps'. In: *Proceedings on Privacy Enhancing Technologies* 2016.3 (July 2016), pp. 24–40. DOI: 10.1515/popets-2016-0014.

[47] K. Rustan M. Leino. 'This is Boogie 2'. June 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[48] Benjamin S. Lerner et al. 'Modeling and Reasoning about DOM Events'. In: *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*. Boston, MA: USENIX, 2012, pp. 1–12. URL: https://www.usenix.org/conference/webapps12/technical-sessions/presentation/lerner.

[49] Farhad Mehta and Tobias Nipkow. 'Proving pointer programs in higher-order logic'. In: *Information and Computation* (2005). 19th International Conference on Automated Deduction (CADE-19), pp. 200–227. ISSN: 0890-5401. DOI: 10.1016/j.ic.2004.10.007.

[50] Pedro J. Molina. 'Quid: Prototyping Web Components on the Web'. In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS 19. Valencia, Spain: ACM, 2019, 3:1–3:5. ISBN: 978-1-4503-6745-5. DOI: 10.1145/3319499.3330294.

[51] MozilJJKKla. *Firefox - Protect your life online with privacy-first products*. 23rd Sept. 2002. URL: https://web.archive.org/web/20191003155535/https://web-platform-tests.org/ (visited on 14/11/2019).

[52] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. *Isabelle/HOL–A Proof Assistant for Higher-Order Logic*. 2002. DOI: 10.1007/3-540-45949-9.

[53] *npm | build amazing things*. 2nd Oct. 2019. URL: https://www.npmjs.com/.

[54] Peter W. O'Hearn, John C. Reynolds and Hongseok Yang. 'Local Reasoning about Programs That Alter Data Structures'. In: *Proceedings of the 15th International Workshop on Computer Science Logic*. CSL '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 1–19. ISBN: 3540425543.

[55] J. Oh, W. H. Ahn and T. Kim. 'Web app restructuring based on shadow DOMs to improve maintainability'. In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. Nov. 2017, pp. 118–122. DOI: 10.1109/ICSESS.2017.8342877.

[56] Claus Pahl. 'A Formal Composition and Interaction Model for a Web Component Platform'. In: *Electronic Notes in Theoretical Computer Science* 66.4 (2002). Formal Methods and Component Interaction (ICALP 2002 Satellite Workshop), pp. 67–81. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80450-X. URL: http://www.sciencedirect.com/science/article/pii/S157106610480450X.

[57] C. Park et al. 'Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T)'. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 552–562. DOI: 10.1109/ASE.2015.27.

[58] Arnd Poetzsch-Heffter, Kathrin Geilmann and Jan Schäfer. 'Infering Ownership Types for Encapsulated Object-Oriented Program Components'. In: *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*. Ed. by Thomas Reps, Mooly Sagiv and Jörg Bauer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 120–144. ISBN: 978-3-540-71322-7. DOI: 10.1007/978-3-540-71322-7_6.

[59] Azalea Raad, José Fragoso Santos and Philippa Gardner. 'DOM: Specification and Client Reasoning'. In: *Programming Languages and Systems*. Ed. by Atsushi Igarashi. Cham: Springer International Publishing, 2016, pp. 401–422. ISBN: 978-3-319-47958-3.

[60] T. Ringer et al. *QED at Large: A Survey of Engineering of Formally Verified Software*. now, 2019. URL: https://ieeexplore.ieee.org/document/8824174.

[61] P. De Ryck et al. 'Protected Web Components: Hiding Sensitive Information in the Shadows'. In: *IT Professional* 17.1 (Jan. 2015), pp. 36–43. ISSN: 1941-045X. DOI: 10.1109/MITP.2015.12.

[62] Gareth David Smith. 'Local Reasoning about Web Programs'. PhD thesis. Imperial College London, 2nd June 2011.

[63] Christian Sternagel and Renè Thiemann. 'XML'. In: *Archive of Formal Proofs* (Oct. 2014). URL: http://isa-afp.org/entries/XML.shtml.

[64] Harvey Tuch, Gerwin Klein and Michael Norrish. 'Types, Bytes, and Separation Logic'. In: *SIGPLAN Not.* (Jan. 2007). ISSN: 0362-1340. DOI: 10.1145/1190215.1190234.

[65] Thomas Tuerk. 'A Formalisation of Smallfoot in HOL'. In: *Theorem Proving in Higher Order Logics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 469–484. ISBN: 978-3-642-03359-9.

[66] Steven Van Acker et al. 'WebJail: Least-privilege Integration of Third-party Components in Web Mashups'. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC 11. Orlando, Florida, USA: ACM, 2011, pp. 307–316. ISBN: 978-1-4503-0672-0. DOI: 10.1145/2076732.2076775.

[67] W3C. *Memorandum of Understanding Between W3C and WHATWG*. 28th May 2019. URL: https://web.archive.org/web/20190702025735/https://www.w3.org/2019/04/WHATWG-W3C-MOU.html (visited on 02/07/2019).

[68] W3C. *web-platform-tests documentation — web-platform-tests documentation*. 2019. URL: https://web.archive.org/web/20191003155535/https://web-platform-tests.org/ (visited on 03/10/2019).

[69] Helen J. Wang et al. 'The Multi-principal OS Construction of the Gazelle Web Browser'. In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM09. Montreal, Canada: USENIX Association, 2009, pp. 417–432.

[70] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. 9th June 2019. URL: `https://web.archive.org/web/20191111164555/https://isabelle.in.tum.de/doc/isar-ref.pdf` (visited on 11/11/2019).

[71] WHATWG. *DOM – Living Standard*. Last Updated 11 February 2019. 11th Feb. 2019. URL: `https://web.archive.org/web/20191003131315/https://dom.spec.whatwg.org/commit-snapshots/7fa83673430f767d329406d0aed901f296332216/` (visited on 03/10/2019).