DOM:

Towards a Formal Specification

Mark Wheelhouse

MSci Project

June 13, 2007

Abstract

We present an initial attempt at providing a formal specification for a minimal structural subsection of DOM Core Level 1. From this 'Minimal DOM' we show how to extended our specification to cover all of the structural behaviour of DOM Core Level 1 and more. We first provide a local specification of our 'Minimal DOM' and then show how the use of Context Logic gives us a powerful and expressive reasoning framework with which to take our local specification to a high level specification. This will be the first time that Context Logic has been applied to a large, real-world, example. Our hope is that our work will demonstrate that Context Logic is just as applicable to real problems as its close sibling Separation Logic.

Supervisor:
Dr. Philippa Gardner

Second Marker:
Ian Hodkinson

Department of Computing
Imperial College London

Acknowledgements

First and foremost I would like to thank Philippa Gardner for agreeing to be my supervisor. She has been a great inspiration throughout the course of the whole project, providing both words of wisdom and a kick in the right direction whenever I needed it.

Secondly I would like to give special thanks to Gareth Smith and Uri Zarfaty for their seemingly endless patience and willingness to make time to help. Without their opinions and assistance this report would have met but a ghost of its true potential.

Finally I would like to thank Ian Hodkinson for his input, especially for his insights into providing examples that make this report clearer for those who do not specialise in logic.

Contents

1 Background **6**
- 1.1 Introduction . . . 6
- 1.2 What is DOM? . . . 6
- 1.3 Why Context Logic? . . . 9
- 1.4 Minimal DOM . . . 11
  - 1.4.1 Node . . . 12
  - 1.4.2 NodeList . . . 13
  - 1.4.3 Commands . . . 13
- 1.5 Data Structure . . . 14
  - 1.5.1 Previous Work - Node level . . . 14
  - 1.5.2 Pure Trees . . . 17
  - 1.5.3 Final Structure . . . 18
  - 1.5.4 Context Application . . . 19
- 1.6 Goals of the Project . . . 20

2 Operational Semantics **22**
- 2.1 Simple Operational Semantics Example . . . 22
- 2.2 The Program State . . . 23
- 2.3 Rules for Append . . . 23
- 2.4 The Commands . . . 26
- 2.5 Evaluation of Operational Semantics . . . 29

3 The Logic **30**
- 3.1 Formulae . . . 30
- 3.2 Classical Formulae . . . 30
- 3.3 Structural Formulae . . . 34
- 3.4 Logic Examples . . . 37

4 Hoare Reasoning **41**
- 4.1 The Reasoning Framework . . . 41
  - 4.1.1 Hoare Rules . . . 43
  - 4.1.2 Local Commands . . . 44
- 4.2 Formulating DOM . . . 45
  - 4.2.1 Append Axiom . . . 46
  - 4.2.2 Axioms . . . 46
  - 4.2.3 Use of Notation Overload . . . 47
  - 4.2.4 Axiom Evaluation . . . 47
  - 4.2.5 Soundness Theorem: . . . 48

5 Deriving the Weakest Pre-conditions **49**
5.1 Notation and Proof Structure 49
5.2 append - a Detailed Proof 50
5.3 The Derivations 51
5.3.1 createNode 51
5.3.2 getNodeName 51
5.3.3 getChildNodes 52
5.3.4 getParentNode 52
5.3.5 insertBefore 53
5.3.6 removeChild 53
5.3.7 getLength 54
5.3.8 getItem 54

6 Building Simple Programs **55**
6.1 replaceChild 55
6.2 cloneNode 56
6.3 hasChildNodes 57
6.4 DOM Core Level 1 - Evaluation 59

7 Going Further **60**
7.1 insertAfter 60
7.2 Equality 61

8 Evaluation **63**
8.1 Success of the Project 63
8.1.1 Pleasing the DOM Community 63
8.1.2 Pleasing the W3C Community 64
8.1.3 The Data Structure 64
8.1.4 The Logical Framework 65
8.1.5 The Axioms 66
8.1.6 Conclusion 66
8.2 Future Work 66
8.2.1 Verification Tool 67
8.2.2 Specifying More of DOM 67
8.2.3 Recursion 67
8.2.4 Small Axioms 68
8.2.5 Multi-holed Contexts 68
8.2.6 Concurrent DOM 68

9 References **69**

10 Appendix I - Constructing Minimal DOM **70**
10.1 Node 70
10.2 NodeList 71
10.3 Constructors 72
10.4 Commands 72

1 Background

1.1 Introduction

In the modern industrial-sector programming-environment companies spend huge amounts of time and money testing and correcting the programs that they produce. The use of formal verification languages, capable of automated code verification against a specification, is rare and still in its infancy, but one can see that the time and money saved would be significant in a world where every minute and every pound seem to matter so much. In order for these verification programs to make it into mainstream use their utility and applicability need to be shown on real programs.

Behind every verification tool there must be a logic capable of expressing program properties and states, then checking these against a suitably formulated specification. During this project we have been looking at the use of Context Logic, a relatively new development in the field, for just this purpose. Context Logic has been proved correct and sound, and has also been shown to work on several toy example languages, but it has yet to be applied to a large, real-world, language. We take DOM, a widely used and well established XML update languages with a rather loose specification, and unite it with Context Logic. Our aim is to both prove the full utility of Context Logic and to bring formal verification to a language that has no nope of it in its current state.

1.2 What is DOM?

DOM, or the Document Object Model, is one of the most widespread XML update languages in use on the Internet. It is a World Wide Wed Consortium (more commonly referred to as W3C) specification of a library for manipulating XML documents. DOM attempts to provide a platform and language neutral interface that allows programs and scripts to dynamically access and update the content, style and structure of XML documents. The idea is that whatever kind of language you are using to modify your XML code, be it imperative, functional, object orientated or any other, the DOM interface at the heart of the code will be the same. This enables a very high degree of portability between platforms which is a valuable property on the World Wide Web.

DOM closely resembles the structure of the document it represents. Each document is represented as a hierarchy of Node objects that are capable of implementing other, more specialised, interfaces. So for example a Text Node would contain textual information, whilst an Attribute Node would contain the attribute value for a list, table or other XML object, but both would be interfaces on top of the same basic structural Node object. The overall DOM keeps track of an unordered set of all the document trees.

DOM Structure Example:

The XML code,

```
<TABLE>
  <TBODY>
    <TR>
      <TD>Name</TD>
      <TD>Project</TD>
    </TR>
    <TR>
      <TD>Mark Wheelhouse</TD>
      <TD>DOM</TD>
    </TR>
  </TBODY>
</TABLE>
```

is represented by the DOM tree in Figure 1.

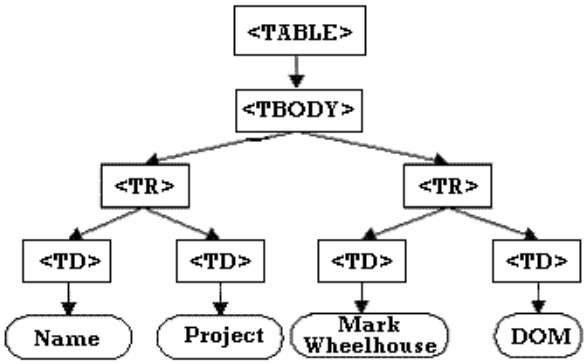

Figure 1: DOM Tree Structure

An interesting feature of DOM is that it is a live update language. This means that an update to one reference of a Node will enforce silent update on all references of that Node without further input from the user. That is, if two processes hold a reference to an object, as soon as one process modifies that object the second process will now be operating upon this modified object.

As we progress through the different sections of this report we shall carry with us a single example, the **append()** command, to show how each stage of our work helps us to progress towards our goal of a more formal specification of DOM.

The append command is called upon a node, lets call this parent, and takes as an argument a target node. This target node, lets call it newChild, is then cut out of its current place in the tree and is added to the end of the parent's list of children. Figure 2 shows how this can be viewed.

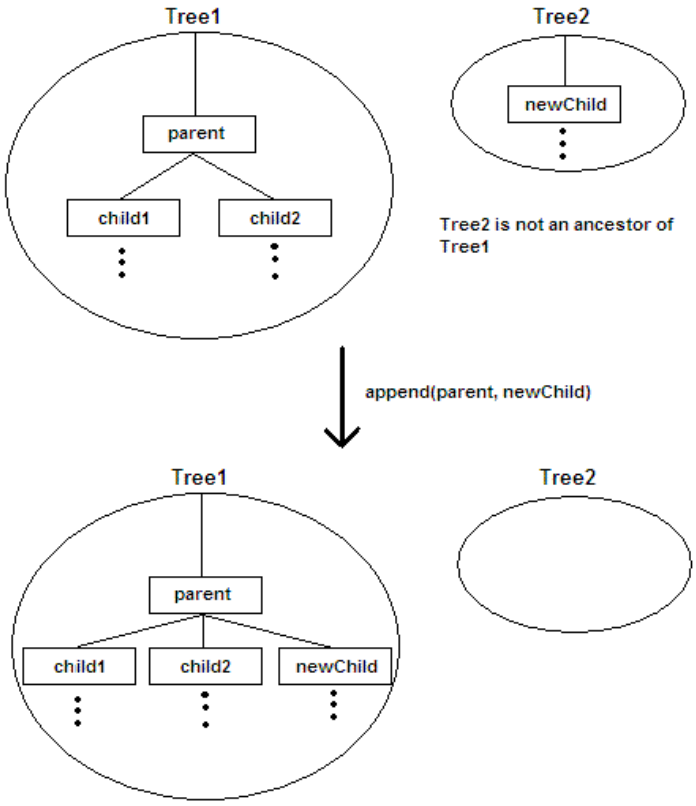

Figure 2: Append Example

Note that we require our target node, newChild, to not be an ancestor of the parent node. If we try to move a node which is an ancestor of the node parent, then the command will return an exception instead of executing. This is a necessary feature of DOM to prevent the user from accidently cutting off parts of a document tree so that they become unreachable. To understand this fully consider the example in figure 3,

This requirement that the Node to be moved must not be an ancestor of its destination will cause us to make some tricky decisions later in our work when

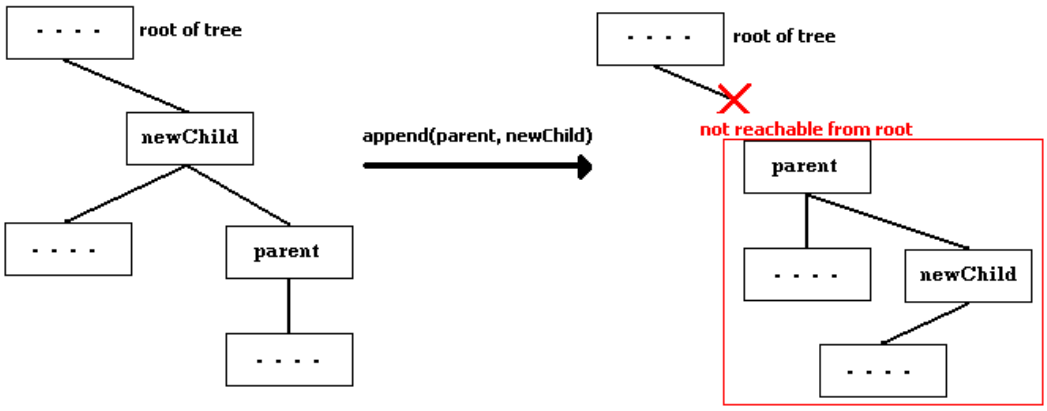

Figure 3: Cutting off a branch

trying to keep the specifications of our commands local. More on this later.

1.3 Why Context Logic?

Context Logic has already been proved to be correct [3] and has been applied to a number of toy examples [3, 9], now it is time to show that Context Logic can be applied to real-world situations.

The current specification of DOM is written using a mixture of IDL (Interface Definition Language), which is an international standard for describing the interface of an Object Model, and a lot of English. The tree structure of the XML is specified with the object structure of an arbitrary object oriented programming language in mind. The operations over this data structure are specified as methods over these objects.

There are two main issues with the current approach.
Firstly, the English language is imprecise and verbose, and therefore so is the specification. DOM has had the benefit of being in use for several years, and there is a large group of volunteers who regularly look for bugs, inconsistencies and undesirable behaviour in DOM, so the specification has grown more accurate over time. However, the still imprecise nature of the specification means that a DOM implementation cannot be precisely verified against the specification.
Secondly, the fact that DOM is written with an arbitrary object orientated programming language in mind somewhat defeats the communities own aim of being platform free. Using object methods to carry out operations prevents the

language from being truly platform free as some languages cannot carry out this kind of parameter passing strategy.

Specification Example:

Looking at the current specification for the append command we find the following specification:

> **appendChild**
> *Adds the node newChild to the end of the list of children of this node. If the newChild is already in the tree, it is first removed.*
>
> **Parameters**
> *newChild The node to add.*
> *If it is a DocumentFragment object, the entire contents of the document fragment are moved into the child list of this node.*
>
> **Return Value**
> *The node added.*
>
> **Exceptions**
> *DOMException*
> *HIERARCHY_REQUEST_ERR: Raised if this node is of a type that does not allow children of the type of the newChild node, or if the node to append is one of this node's ancestors.*
>
> *WRONG_DOCUMENT_ERR: Raised if newChild was created from a different document than the one that created this node.*
>
> *NO_MODIFICATION_ALLOWED_ERR: Raised if this node is readonly.*

This specification tells us roughly how the append command works, and as far as English specifications go it is quite precise, but we can never hope to use such a specification as part of a formal verification system.
The one specific issue with this specification is working out what happens in the case where the target child to append is an ancestor of the target parent node. We know that this case results in a DOMException being thrown, but where is this in this specification? If you look long and hard you'll find it buried at the end of the definition of HIERARCHY_REQUEST_ERR. We mention this as it took our group several attempts to locate this case. That it has fooled several intelligent minds leads us to conclude that the specification must be poorly laid out. In fact we feel that this exceptional case really ought to be specified separately as it is very significant in practice.

It turns out that this type of criticism is not rare in commands of the DOM specification. Often the special cases where the commands do not behave as expected are left out or not fully documented. This is in part due to the iterative

nature in which DOM has been built up over the years where exceptional cases and unexpected behaviour are only documented once they have been discovered.

The append command above takes just a single input, the child in question, but throughout the rest of this paper we shall refer to an append command that takes two inputs, the parent node and the child node. This is because the append command in DOM is called as a method upon the parent node, and so the parent is passed as an implicit argument. We choose to make this implicit argument passing explicit for the duration of our work, as we feel this leads to a clearer understanding of our work.

Context Logic is a spatial logic for structurally reasoning about data. A Context Logic for trees will provide a more precise language with which to specify a library like DOM. We want to use Context Logic to give a less ambiguous, more formal specification for DOM that can be used to verify DOM implementations, hopefully in an automated fashion (although that is outside the scope of this project). In doing so we must be careful not to discard any of DOM's aimed for platform free nature. Creating this specification is a true test of the expressive power of Context Logic.

By using Context logic will will address our two main criticisms of the current DOM specification. Namely, by eliminating the English language from the specification we will remove countless occurrences of ambiguity and greatly increase the precision of the specification. The new specification aims to be formal and precise enough to allow formal verification tools to be built. Also, we will attempt to move away from the solely object orientated viewpoint of the current DOM specification so that we provide a specification which is truly platform independent as the DOM community originally intended. However, at this stage we will not be attempting to tackle the entirety of DOM in one go, but we shall concentrate on a suitable subsection of DOM. We will take care to work in such a fashion that our work can be scaled up to the full DOM at a later point, but complete DOM is too large for the scope of this project. Also, at this stage, we are not in a position to provide a verification tool. It is our hope that the results of this project will allow the easy construction of a formal verification tool for the subsection of DOM that we are going to consider, but again, time pressures prevent us from including that work in the scope of this project.

1.4 Minimal DOM

The entirety of DOM is far too large to consider in one go, so we have decided to begin our specification looking at just the core parts of DOM. Specifically we are restricting our investigation to DOM Core Level 1 [12], a minimal set of objects and interfaces for accessing and manipulating document objects. The

Core functionality specified should be sufficient to allow software developers and web script authors to access and manipulate parsed XML content inside all conforming products.

In fact, even DOM Core Level 1 is more expressive than we desire as a staring point. Choosing to restrict ourselves to structural considerations only, we can create a smaller minimal subsection of DOM. We shall refer to this henceforth as Minimal DOM. We will fully specify this minimal subsection and show that the remaining behaviour of DOM Core Level 1 can be derived from it. Further we hope to show that higher levels of DOM may also be derived from this minimal subset with a minimal number of extensions made to encompass the new concepts in these higher levels of DOM.

In the following section we shall summarise what we have identified as Minimal DOM. A full and detailed discussion of our reasoning can be found in 'Appendix I - Constructing Minimal DOM', but we just present our results here.

1.4.1 Node

The principle data-type we choose to deal with is the Node element. This is a single element of the DOM tree which can be viewed as in figure 4,

Figure 4: Minimal DOM Node

Each Node has the following attributes:

id: a *unique* identifier for the Node. This can be thought of as a memory location that we can point to.

tag: a flat representation of the data associated with the Node.

fid: a *unique* identifer for the list of children of the Node.

forest: an ordered list of pointers to other Nodes, the children of this Node.

1.4.2 NodeList

We choose to use a separate data-type for the ordered list of nodes. This NodeList can be viewed as in figure 5

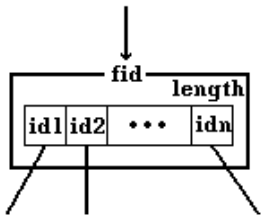

Figure 5: Minimal DOM NodeList

Each NodeList has the following attributes:

fid: a *unique* identifier for the NodeList. Just as for Nodes, this can be thought of as a memory location that we point to.

length: the number of items in the list is tracked explicitly.

For convenience we assume that our lists are 0 indexed, that is that the first element is at position 0 of the list.

1.4.3 Commands

In Appendix I we give a detailed discussion of which commands we are choosing to work with. Given below is the full list of commands that we shall be using in our specification along with a brief English description, obtained from the existing DOM specification, of the behaviour of each command.

1. createNode(String: tag)
   creates a new Node with fresh id's and its tag (the flat representation of the data associated with this new Node) is set to the input string

2. getNodeName(Node: node)
   returns the sting stored in the tag of the Node with the input identifier

3. getChildNodes(Node: parent)
   returns the identifier of the NodeList of children for the Node with the input identifier

4. getParentNode(Node: child)
   returns the parent Node of the Node with the input identifier

5. insertBefore(Node: parent, Node: newChild, Node: ref)
   inserts the Node with identifier newChild before the Node with identifier ref in the NodeList associated with the Node with identifier parent

6. removeChild(Node: parent, Node: child)
   removes the Node with identifier child to the top level of the DOM tree from the NodeList of the Node with identifier parent and returns the Node identifier for the removed child

7. append(Node: parent, Node: newChild)
   puts the Node with identifier newChild at the end of the NodeList of the Node with identifier parent

8. getLength(NodeList: fid)
   returns the length of the NodeList with identifier fid

9. getItem(Int: length, NodeList: fid)
   returns the identifier of the Node at position length of the NodeList with identifier fid

1.5 Data Structure

The next step in specifying DOM is to define the data structure that our commands will operate over. The current DOM specification makes no attempt to give an explicit data structure, leaving it to the user to think in whatever style seems natural to themselves. Clearly, if we are to produce a more formal specification, we must provide the definitions of this data structure. However, in doing so we must be careful to avoid making any assumptions that will inhibit the platform free nature of DOM implementations. The following section details the data structures which were considered during the course of this project, the reasons for making our decisions and the final data structure that was settled upon.

1.5.1 Previous Work - Node level

The logical place to start our search for a suitable data structure is to begin by looking at the work of Cheung [6]. Cheung decided to aim his reasoning at a low level, concentrating on the structure of the DOM tree from the point of view of the Nodes . The standard Node element which he designed appears in figure 6.

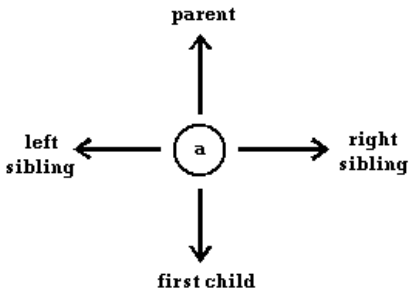

Figure 6: Cheung Node

From this basic element, Cheung defined a set of Node level commands. When combined together these provide a similar functionality to the commands which we have chosen to consider. Cheung chose to operate over the data structure as defined below:

forest ϕ ::=	$\emptyset$	*empty/null element*
	$a[\phi] \mid \phi$	*node composition*
forest segment c ::=	$\_$	*context hole*
	$a[\phi] \mid c$	*right sibling context*
	$a[c] \mid \phi$	*child context*

The structure of the child nodes is ordered, with the composition operator '|' being associative and non-commutative with identity $\emptyset$.

We found that this data structure was not suitable for our intended purpose.
Firstly, we want to carry out our reasoning at the high level, linking to established low level reasoning at a later point. Cheung's work found itself having to make some very tricky decisions due to low level complications that we will not be confronted with at the high level. Some of these decisions may actually make it harder for us, at the high level, rather than easier.
Secondly, we want to keep the footprints of our commands as simple as possible. The footprint of a command is the portion of the data structure that is required or affected by the commands. With a sibling approach to the structure of the DOM tree the footprint is the complex combination of the Node the method is called upon and the Node the method returns, if it exists. Picking out these two chunks of the DOM tree is not a simple thing to specify. We would prefer a neater solution.
Finally, Cheung's data structure has a severe limitation in the construction of contexts. It is not possible to place a context hole at any location in our data structure, we may only have holes to the right of a Node or below a Node. To

clarify this limitation let us consider some examples.

Context Hole Limitations:

The context definition allows us to construct (1): $a[\emptyset] \mid b[\emptyset] \mid \_$
and (2): $a[\emptyset] \mid b[\_] \mid c[\emptyset]$, see figure 7, where the context hole occurs at the rightmost sibling or first child respectively.

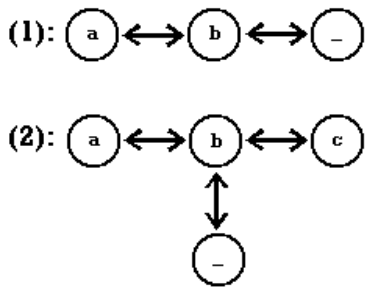

Figure 7: We can construct these

However we cannot construct (3): $a[\emptyset] \mid \_ \mid c[\emptyset]$ or (4): $\_ \mid b[\emptyset] \mid c[\emptyset]$, see figure 8, where the context hole appears in a more arbitrary position of the tree.

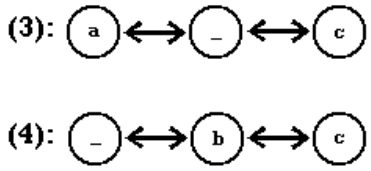

Figure 8: We can't construct these

The work of Kearns [7] shows a possible solution to this limitation by introducing a left adjunct $c \mid a[\phi]$ of the sibling context definition. This allows the placement of a context hole at any point in the tree. However, there are a number of complex subtleties that arise from this seemingly trivial extension of the data structure. Rather than grappling with this increased complexity when making use of contexts, we instead choose to maintain our aim of finding a simpler data structure and turn our attention to higher level data structures where these subtleties no longer concern us.

1.5.2 Pure Trees

An initial suggestion of a way to move to a higher level model was to consider using an ordinary tree structure similar to that found in [3], the idea being that keeping the data structure as simple and general as possible would also make the specification simple and general, and therefore more portable across different platforms. The suggested structure was defined intuitively as follows:

$$
\begin{array}{rcl}
\text{Tree } T ::= & \emptyset & \textit{empty/null tree} \\
& n_{\text{tag}}[T] & \textit{tree node} \\
& T \mid T & \textit{parallel tree composition} \\
\\
\text{Tree Context } C ::= & \_ & \textit{context hole} \\
& n_{\text{tag}}[C] & \textit{subtree context} \\
& T \mid C & \textit{right context} \\
& C \mid T & \textit{left context}
\end{array}
$$

Here $\emptyset$ represents the null tree, n represents the Node identifier and 'tag' is the flat representation of the data contained within the Node. The list of children is an ordered and indexed list and the context definition allows a list of trees containing a context hole at any location in the list.

This representation, however, also turned out to not represent all of the features of the DOM data structure. Specifically, it is still not possible for us to represent the top level of the DOM data structure, namely that DOM stores an *unordered* set of trees for different documents. We can extend the definition of our Tree with some commutative parallel composition operator for trees, but this would have to be restricted to use at the top level only. It would be very cumbersome indeed to have to carry around some notion of the depth of the tree we are working at and we would loose any hope of keeping our reasoning local to the subtree we are operating over, as we would always have to be counting our depth from the root of the tree. Clearly, we need to search for a more specific data structure than can be given by a single data data type, and its context analogue, alone.

1.5.3 Final Structure

Finally, we turned our attention to a multi-typed data structure. Starting from the work of Smith [8] we constructed a Grove, Tree and Forest data structure. Smith proposed that instead of trying to use a single data type that we instead use three. Namely that we have a Grove of trees at the top level of the DOM data structure, and a hierarchy of Trees containing Forests of Trees beneath this. After many discussions together, we arrived at the following data structure

Grove $G ::=$	$\emptyset$	*empty grove (empty set)*
	$G \oplus G$	*grove composition*
	T	*tree*
Tree $T ::=$	$\text{tag}_{\text{id}}[F]_{\text{fid}}$	*tree node - no empty tree*
Forest $F ::=$	$\emptyset$	*empty forest (empty list)*
	$F \times F$	*forest composition*
	T	*tree*
Grove Context $CG ::=$	$\_$	*context hole*
	$CG \oplus G$	*sibling context*
	CT	*tree context*
Tree Context $CT ::=$	$\text{tag}_{\text{id}}[CF]_{\text{fid}}$	*subtree context*
Forest Context $CF ::=$	$\_$	*context hole*
	$CF \times F$	*left context*
	$F \times CF$	*right context*
	CT	*tree context*

As before 'tag' is the flat representation of the data contained within the Node, but we now have an 'id' representing the Node identifier and 'fid' representing the identifier for the forest of children. We choose to identify the forest of children within a Node explicitly as this is closer to the current DOM specification. The $\oplus$ operation used above is addition over sets and hence is both associative and commutative. The $\times$ operation used above is list concatenation and hence is associative, but is not commutative. Both treat $\emptyset$ as their identity element. This data structure gives us the ability to place our empty context anywhere in the tree without the need for complex context predicates and is therefore a significant improvement upon the previous attempts.
The biggest change made to [8] was to overload the $\emptyset$ element. It now represents

both the empty tree and the empty list. The reason for this significant change is to remove the empty tree element. An empty tree is a difficult to describe object. Whist it is commonly understood what the empty list ,[ ], represents, what is meant by the empty tree? It is possible to think of this as a pointer to some null object that can occurs whenever an empty tree is referred to, but this seems a bit contrived. It is also possible to think of a reference to an empty tree as a dangling pointer, that is a pointer that points nowhere. However this notion will be confusing when we begin to apply context logic to our data structure, as a notion of dangling pointers already exists here. Since both empty sets and empty lists are well understood and commonplace concepts which are easy to explain it seems natural to use them instead.
In practice it is always obvious from our program context wether the empty set or empty list is being referred to and the notational benefits from overloading are substantial as we shall see later.

1.5.4 Context Application

The use of Contexts above leads us to define a context application function ap(context, data) which will define how the different contexts are applied to the different data types. However to do this we in fact need to define a number of sub-functions for each type of context application.

$$
\begin{array}{rcl}
\mathrm{ap}_{g\times g}(\_, G) & \to & G \\
\mathrm{ap}_{g\times g}(CG \oplus G', G) & \to & \mathrm{ap}_{g\times g}(CG, G) \oplus G' \\
 & & \\
\mathrm{ap}_{g\times t}(\_, T) & \to & T \\
\mathrm{ap}_{g\times t}(CG \oplus G, T) & \to & \mathrm{ap}_{g\times t}(CG, T) \oplus G \\
\mathrm{ap}_{g\times t}(CT, T) & \to & \mathrm{ap}_{t\times t}(CT, T) \\
 & & \\
\mathrm{ap}_{g\times f}(CG \oplus G, F) & \to & \mathrm{ap}_{g\times f}(CG, F) \oplus G \\
\mathrm{ap}_{g\times f}(CT, F) & \to & \mathrm{ap}_{t\times f}(CT, F) \\
 & & \\
\mathrm{ap}_{f\times t}(\_, T) & \to & T \\
\mathrm{ap}_{f\times t}(CF \times F, T) & \to & \mathrm{ap}_{f\times t}(CF, T) \times F \\
\mathrm{ap}_{f\times t}(F \times CF, T) & \to & F \times \mathrm{ap}_{f\times t}(CF, T) \\
\mathrm{ap}_{f\times t}(CT, T) & \to & \mathrm{ap}_{t\times t}(CT, T) \\
 & & \\
\mathrm{ap}_{f\times f}(\_, F) & \to & F \\
\mathrm{ap}_{f\times f}(CF \times F', F) & \to & \mathrm{ap}_{f\times f}(CF, F) \times F' \\
\mathrm{ap}_{f\times f}(F' \times CF, F) & \to & F' \times \mathrm{ap}_{f\times f}(CF, F) \\
\mathrm{ap}_{f\times f}(CT, F) & \to & \mathrm{ap}_{t\times f}(CT, F) \\
 & & \\
\mathrm{ap}_{t\times t}(\mathrm{tag}_{\mathrm{id}}[CF]_{\mathrm{fid}}, T) & \to & \mathrm{tag}_{\mathrm{id}}[\mathrm{ap}_{f\times t}(CF, T)]_{\mathrm{fid}} \\
 & & \\
\mathrm{ap}_{t\times f}(\mathrm{tag}_{\mathrm{id}}[CF]_{\mathrm{fid}}, F) & \to & \mathrm{tag}_{\mathrm{id}}[\mathrm{ap}_{f\times f}(CF, F)]_{\mathrm{fid}}
\end{array}
$$

Any combinations of contexts and data that have not been expressed above are not legal over our defined data structure. It does not make sense to apply grove data to a forest context, for example, so we shall not define such applications. For the rest of the paper we shall refer only to one application function, ap(), but it will always be obvious, in context, which of the specific application functions is being applied.

1.6 Goals of the Project

The rest of this report follows our work as we build towards a complete specification of Minimal DOM.
In Section 2 we will use the current DOM Specification to produce Operational Semantics for each of the commands of Minimal DOM. Whilst these give a much more formal specification of DOM they are not compositional enough for our

needs and are also very hard to read. They do however provide us with a useful stepping stone in our path to an understandable and complete specification for Minimal DOM.
Section 3 of this report details the Logic that we will be using to describe the pre- and post-conditions of our commands. We give a full definition of the logic along with several examples of how the logic can be used.
In Section 4 we introduce Hoare Reasoning which allows us to use context logic in a powerful way. By specifying the Local Hoare Triples of our minimal commands we can create 'building blocks' of logic which can go on to provide the pre- and post-conditions for bigger and more complicated programs. We give Axioms for each of our commands and discuss locality in more detail.
Section 5 concerns the formal derivations of the weakest pre-conditions for each of our minimal commands. The proofs for each command are given in full along with a discussions of the salient points of our append example.
In section 6 of the report we use our fully specified set of minimal commands to show how we can produce the complete specification for the structural parts of DOM Core Level 1.
Section 7 shows how we can in fact go further than this and fully specify a much larger set of commands.
Finally, section 8 provides a detailed analysis of the project and suggests possible paths that might be followed for future research.

The successful outcome of the end goal of a complete specification for DOM will have several benefits.
Firstly, a formal, unambiguous specification will allow verification of DOM implementations which will be a desirable outcome for the DOM community.
Secondly, the W3C community will have a more rigorous specification which should make the use and future extensions of DOM a much easier task than is currently the case.
Finally, by satisfying the demands of both of these communities we will have demonstrated that Context Logic can be used in real-world situations, not only on toy examples.

2 Operational Semantics

Now that we have provided a formal definition of the data structure underlying DOM Core level 1, we wish to turn our attention to the identified minimal subset of commands. The first step here, in providing a more formal specification, is to define the Operational Semantics of our commands. The behavior of the commands was interpreted from the current English specification and converted into Operational Semantics following standard notation similar to that set out in [15]. For those of you not familiar with Operational Semantics we shall first consider a simple example that illustrates the intended meaning of this notation and then give a detailed explanation of the semantics for the append command.

2.1 Simple Operational Semantics Example

Let us consider the simple command add(x, n) operating over a program state s capable of storing only integer variables. The command simply adds the integer value n to the value stored in variable x. The Operational Semantics for this would look like,

$$\frac{s(x) = m}{\text{add}(x, n),\ s \Rightarrow s[\ x \mapsto [[m + n]]\ ]}$$

The top line of the rule defines any precondition that must be true for the semantics to hold. It is possible for a command to behave differently depending upon its input, this would be represented by the command having multiple rules associated with it, each with a different precondition. If none of the rules for a command have a pre-condition that holds for a given input, then the command will fault on that input.
Everything below the line describes the postcondition of the command. This will be of the form **(command, state ⇒ newstate)** where **state** tells us what the program state looks like before the command is run and **newstate** tells us what the program state looks like after the command has been successfully run. So in the above rule for the add command, the pre-condition tells us that the variable x must exist in the program state s before the command is run. Moreover it tells us that the value stored in x in state s is some integer m.
So the command **add(x, n)** carried out in this state s will evaluate to ⇒ the new state $s[\ x \mapsto [[m + n]]\ ]$
where,
$[[m + n]]$ evaluates the value of the arithmetic expression $m + n$, call this a,
and $s[\ x \mapsto a]$ tells us that the value of variable x in the state s is now a.

Note that if the state s were able to contain other types of variables other than just Integers, then we would need to quantify in the precondition that m is indeed an Integer. Any other case would simply result in a program fault. For the duration of this paper we shall be using typed variables, which implicitly verify that their value is of the same data type as themselves.

2.2 The Program State

Our first step in producing the Operational semantics for our commands is to define the program state that the commands will operate over. We will be using the tuple (s, G), where we have a Grove (unordered set of trees) G, which is just as defined in the previous section and also a stack s in which we will store our variables. It is important to note that we can only store pointers to Grove objects in the stack, and never the objects themselves. DOM does not allow the storage of Nodes or NodeLists within the stack and in order to keep our specification as close to DOM as possible we must also maintain this restriction.

So, we define the stack that the semantics will operate over as a partial function from variables to values:

$$s : (\mathbf{tag} \to \text{tag}) \times (\mathbf{id} \to \text{id}) \times (\mathbf{n} \to n) \times (\mathbf{bool} \to \text{bool})$$

where tag is a String, id is the identifier of an object in the grove, n is a natural number, and bool $\in$ {true, false}.

2.3 Rules for Append

The key to understanding the operational semantics in this section is to understand the **ap(context, data)** function. This function is merely a formal way of representing context application. it tells us that the **data** is placed inside the **context** in place of the context's hole element '_'. So if we have a grove context

$$CG \equiv \text{tag}_{\mathbf{parent}}\ [\_]_{\text{fid}}$$

and we apply it to some data, lets say a node

$$D \equiv \text{tag}'_{\mathbf{child}}\ [\emptyset]_{\text{fid}'}$$

then this node would be put into the context in place of the '_' element to give us the resulting tree

$$ap(CG, D) \equiv \text{tag}_{\mathbf{parent}}\ [\text{tag}'_{\mathbf{child}}\ [\emptyset]_{\text{fid}'}]_{\text{fid}}$$

This can be viewed as shown in figure 9.

Let us now consider the append command in detail. As we discussed before, the append command will only execute correctly so long as the child to be moved is not above the parent node in the tree. However this still leaves four cases of where the child can be located in the tree, each of which has a separate rule associated with it. Figure 10 shows these four cases.

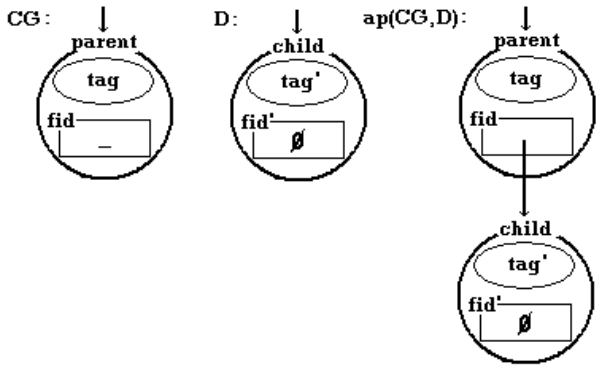

Figure 9: Context Application

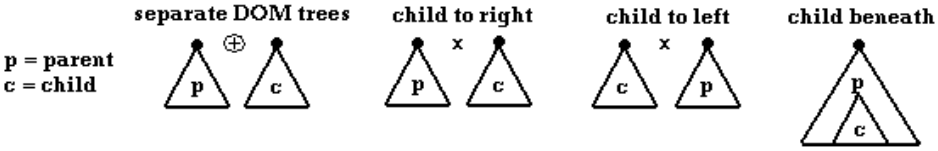

Figure 10: The cases of Append

The first case to consider is where the child and parent occur in completely separate DOM trees, that is that their trees are connected only at the top level. The rule for this case is as follows,

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F]_{\mathrm{fid}}) \oplus \mathrm{ap}(CG'', \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}) \oplus \mathrm{ap}(CG'', \emptyset))) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

The bottom of this rule tells us that the only thing the append command affects is the data structure. Before the command is run we have the grove G, and after it is run we have the grove G'. Above the line we have the definitions of the forms that G and G' must take in order for this rule to hold.
In G the parent and child must be in different groves which are joined by the $\oplus$ operator, so these trees must be joined at the top level. The use of contexts allows these groves to be of any arbitrary size or shape, the only condition is that parent and child must not be in the same tree. Since $\oplus$ is a commutative operator it does not matter which side of the parent the child occurs on as there is no ordering in the top level set of trees.
In the resulting tree G' the child has moved to be at the end of the list of parent's

children. Note also that the child has been removed entirely from the grove it was originally a part of and has been replaced by the $\emptyset$ element. The overloading of notation here saves us from having separate cases for when the child comes from the top level and when it comes from deeper in a tree as $\emptyset$ will be typecast to whichever empty element it needs to be. Note also that no other parts of the overall context grove are affected by the command, it is said to be acting locally.

The second case to consider is where the child occurs somewhere to the right of the parent. In this case we get the rule,

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F]_{\mathrm{fid}}) \times \mathrm{ap}(CG'', \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}) \times \mathrm{ap}(CG'', \emptyset))) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

As before only the data structure is affected by the command. Again we have that parent and child must be in separate trees before the command is called, but these trees now diverge somewhere within a single DOM tree and not at the top level. Since the $\times$ operator is not commutative it is important that the child is on the right of the parent in this case. Also note that the order of the rest of the tree is still preserved after the command has executed. By symmetry the case where the child is on the left works in exactly the same way, so we will not cover it here.

All that we are left to explain is the case where the child occurs in the tree somewhere beneath the parent. The rule in this case is,

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[\mathrm{ap}(CF, \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'})]_{\mathrm{fid}}) \\ G' \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[\mathrm{ap}(CF, \emptyset) \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

The layout of the above rule is somewhat different to that of the other rules for append. We no longer have the composition of two trees, instead we have that the child lies somewhere in the subtree beneath the parent node. So the parent node contains a forest context which contains the child, rather than some arbitrary forest. As with all the append rules, once the command has executed the child has been placed at the end of the list of children of parent, which in this case means it is appended onto the end of a forest context.

Since we do not provide semantics for the case where the child occurs as an ancestor of the parent, this program state would result in an error if it was encountered. Whilst we could explicitly create separate error cases for each for the different error states this adds little to the semantics and is a trivial extension. We instead choose to assume that if a program state is encountered that we do not have operational semantics for this results in a DOMException being returned and the program would halt.

2.4 The Commands

Listed below are the full operational semantics for Minimal DOM as identified above. We have included all four cases of append for reference and we give a brief explanation of each command.

createNode

When we create a new Node in DOM it is created at the top, grove level, of the DOM tree. So our new Node, which has fresh id and fid to preserve uniqueness, is joined to the existing (possibly empty) grove by a $\oplus$ operation. Note that we also store a pointer to the Node in the variable **node**.

$$\frac{G' \equiv G \oplus s(\mathbf{tag})_{\mathrm{id}}[\emptyset]_{\mathrm{fid}} \qquad (\mathrm{id},\mathrm{fid}\ fresh)}{\mathbf{node} = \mathrm{createNode}(\mathbf{tag}),\ s,\ G \Rightarrow s[\mathbf{node} \mapsto \mathrm{id}],\ G'}$$

getNodeName

The get methods of DOM do not change the data structure of the DOM tree, but simply return a value from the tree. In this case we require that some Node with identifier **id** exists in the grove. We return the tag of this Node to the $\mathbf{tag}'$ variable.

$$\frac{G \equiv \mathrm{ap}(CG, \mathrm{tag}_{\mathbf{s(id)}}[F]_{\mathrm{fid}})}{\mathbf{tag}' = \mathrm{getNodeName}(\mathbf{id}),\ s,\ G \Rightarrow s[\mathbf{tag}' \mapsto \mathrm{tag}],\ G}$$

getChildNodes

Much the same as getNodeName, we require that a Node with the correct identifier does exist in the graph and then return its forest identifier **fid** in the **forest** variable.

$$\frac{G \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[F]_{\mathrm{fid}})}{\mathbf{forest} = \mathrm{getChildNodes}(\mathbf{parent}),\ s,\ G \Rightarrow s[\mathbf{forest} \mapsto \mathrm{fid}],\ G}$$

getParentNode

The last of the get methods for Nodes, once again we require existence of the correct Node, and assign the return value to the **parent**' variable.

$$\frac{G \equiv \mathrm{ap}(CG, \mathrm{tag}_{\mathrm{parent}}[F_1 \times \mathrm{tag}'_{s(\mathbf{child})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}})}{\mathbf{parent}' = \mathrm{getParentNode}(\mathbf{child}),\ s,\ G \Rightarrow s[\mathbf{parent}' \mapsto \mathrm{parent}],\ G}$$

insertBefore

This move command is very similar to the append command. The only difference is that the child is inserted before a specific element of the parents list of children, rather than at the end of the list. Just as for the append command, there are four cases for where the parent and child are in relation to one another in the DOM tree. The first case has the parent and child in separate trees in the grove, the second and third cases have the child to the right, or left, of the parent and in the last case the child is in the subtree beneath the parent. The structure of these rules is very similar to those of append, only the last case is significantly different. The condition,

$$(F_1 \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2) = \mathrm{ap}(CF, \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''})$$

simply specifies that the child node is somewhere in the subtree beneath the parent node before the command is executed. It could be in any part of F_1, F_2 or F', but by using the above condition we can avoid writing three more rules for this command. Similarly we use this notation again for the program state after the execution of the command, where the child node has now been removed from its original place in the tree.

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}) \oplus \mathrm{ap}(CG'', \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''} \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}) \oplus \mathrm{ap}(CG'', \emptyset))) \end{array}}{\mathrm{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}),\ s,\ G \Rightarrow s,\ G'}$$

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}) \times \mathrm{ap}(CG'', \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''} \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}) \times \mathrm{ap}(CG'', \emptyset))) \end{array}}{\mathrm{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}),\ s,\ G \Rightarrow s,\ G'}$$

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG'', \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''}) \times \mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG'', \emptyset) \times \mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''} \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}))) \end{array}}{\mathrm{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}),\ s,\ G \Rightarrow s,\ G'}$$

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}'_{\mathrm{ref}}[F]_{\mathrm{fid}'}\ \mathrm{x}F_2]_{\mathrm{fid}}) \\ (F_1 \times \mathrm{tag}'_{s(\mathbf{ref})}[F]_{\mathrm{fid}'} \times F_2) = \mathrm{ap}(CF, \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''}) \\ G' \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[F'_1 \times \mathrm{tag}''_{s(\mathbf{newChild})}[F']_{\mathrm{fid}''} \times \mathrm{tag}'_{s(\mathbf{ref})}[F''']_{\mathrm{fid}'} \times F'_2]_{\mathrm{fid}}) \\ (F'_1 \times \mathrm{tag}'_{\mathrm{ref}}[F''']_{\mathrm{fid}'} \times F'_2) = \mathrm{ap}(CF, \emptyset) \end{array}}{\mathrm{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}),\ s,\ G \Rightarrow s,\ G'}$$

removeChild

DOM never explicitly deletes Nodes form a tree, but assumes that garbage collection will remove all unused Nodes in good time. The only method it does

provide is the removeChild method which allows us to take a single Node out of any part of a DOM tree and move it to the grove level. The precondition here is merely that we have such a child, and the post condition simply states that we indeed pull the child out of the tree, along with its subtree, and add it to the grove level. We also store a pointer to the removed Node in the variable **node** for later use if we wish to refer to this Node again. At this time we are not explicitly modelling garbage collection, so if we do not want to use a Node again we will simply discard the variable that points to it.

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times \mathrm{tag}'_{s(\mathbf{child})}[F]_{\mathrm{fid}'} \times F_2]_{\mathrm{fid}}) \\ G' \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[F_1 \times F_2]_{\mathrm{fid}}) \oplus \mathrm{tag}'_{s(\mathbf{child})}[F]_{\mathrm{fid}'} \end{array}}{\mathbf{node} = \mathrm{removeChild}(\mathbf{parent}, \mathbf{child}),\ s,\ G \Rightarrow s[\mathbf{node} \mapsto s(\mathbf{child})\ ],\ G'}$$

append

We have already discussed this command in detail at the beginning of this section. Below we have all four rules for this command given together.

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F]_{\mathrm{fid}}) \oplus \mathrm{ap}(CG'', \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}) \oplus \mathrm{ap}(CG'', \emptyset))) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F]_{\mathrm{fid}}) \times \mathrm{ap}(CG'', \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}) \times \mathrm{ap}(CG'', \emptyset))) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG'', \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}) \times \mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F]_{\mathrm{fid}}))) \\ G' \equiv \mathrm{ap}(CG, (\mathrm{ap}(CG'', \emptyset) \times \mathrm{ap}(CG', \mathrm{tag}_{s(\mathbf{parent})}[F \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}))) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[\mathrm{ap}(CF, \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'})]_{\mathrm{fid}}) \\ G' \equiv \mathrm{ap}(CG, \mathrm{tag}_{s(\mathbf{parent})}[\mathrm{ap}(CF, \emptyset) \times \mathrm{tag}'_{s(\mathbf{newChild})}[F']_{\mathrm{fid}'}]_{\mathrm{fid}}) \end{array}}{\mathrm{append}(\mathbf{parent}, \mathbf{newChild}),\ s,\ G \Rightarrow s,\ G'}$$

getLength

The getLength command returns the length of a forest of children in variable **length**. This get method makes no modification to the grove, but does require that a forest with identifier **fid** exists somewhere in the DOM tree. There are 2 cases for this command, one where the list is non-empty and one where the list is empty. Although '...' is not a formally defined concept, it is sufficient for our

purposes here. If the Operational Semantics were to be our final product this concept would need to be more rigourously defined.

$$\frac{G \equiv \mathrm{ap}(CG, \mathrm{tag}_{\mathrm{id}}[T_1 \times ... \times T_n]_{s(\mathbf{fid})})}{\mathbf{length} = \mathrm{getLength}(\mathbf{fid}),\ s,\ G \Rightarrow s[\mathbf{length} \mapsto n],\ G}$$

$$\frac{G \equiv \mathrm{ap}(CG, \mathrm{tag}_{\mathrm{id}}[\emptyset]_{s(\mathbf{fid})})}{\mathbf{length} = \mathrm{getLength}(\mathbf{fid}),\ s,\ G \Rightarrow s[\mathbf{length} \mapsto 0],\ G}$$

getItem

The last command we consider, getItem, is similar to the other get methods. It does not modify the grove, it returns to variable **node** and it requires the existence of the Node it is asked for. Note that we also require the integer input to be within the list bounds.

$$\frac{\begin{array}{c} G \equiv \mathrm{ap}(CG, \mathrm{tag}_{\mathrm{id}}[T_0 \times ... \times T_n]_{s(\mathbf{fid})}) \\ T_{s(\mathbf{int})} = \mathrm{tag}'_{\mathrm{id}'}[F]_{\mathrm{fid}'} \qquad (0 \leq s(\mathbf{int}) \leq n) \end{array}}{\mathbf{node} = \mathrm{getItem}(\mathbf{fid}, \mathbf{int}),\ s,\ G \Rightarrow s[\mathbf{node} \mapsto \mathrm{id}'],\ G}$$

2.5 Evaluation of Operational Semantics

Now that we have given the operational semantics for Minimal DOM we are part of the way towards our final goal. Hopefully you can now see that the Operational semantics are not that complicated to understand and do indeed give a much tighter specification of Minimal DOM than the existing specification. However, despite the fact that we have added accuracy and precision using this form of specification, there is nothing new here. All of this behaviour could be retrieved from the existing specification, though it might take a bit more decoding and intuition than we would prefer. We would have to write entirely new Operational semantics for new commands as this style of specification does not lend itself to easy composition. We turn our attention to logical languages, rather than this very computation driven approach so that we may produce an easily extendable specification that we could later hope to expand to cover the whole of DOM. The use of logic will also lead to more succinct pre- and post-conditions for our commands without the need for multiple rules for each of the more complicated move commands. We appreciate that the operational semantics are not easy for the untrained eye to read and this provides another motivation for us to move onto the use of logical languages to provide a specification for Minimal DOM.

3 The Logic

The Operational Semantics defined above, whilst giving us a precise definition of each command, are not flexible enough to provide us with a method of DOM implementation verification. Since we have to define new Operational semantics for every new command we add, and there is no compositional method for doing this, we choose to move on to the use of logic to express the program state and its properties. In order to provide a strong formalisation of our model we need to be able to express formulae describing the properties of the Groves, Trees and Forests defined in our data structure.

3.1 Formulae

We will introduce the formulae for our logic in two stages. First, we shall be looking at the classical logic formulae which operate over our data structures. Secondly, we will be describing the formulae that give the structure of our program states and the contexts that we shall be using. As well as defining these formulae will shall also be giving their satisfaction semantics, ie. what it means for a given data structure to satisfy a logical formula of each type. Finally, we shall construct a number of examples from our fully defined logic, concentrating on the formalisms that we shall be utilising in the remaining parts of this report.

In our Logic we define the following types of formulae,

Grove Data	PG
Tree Data	PT
Forest Data	PF
Grove Contexts	KG
Tree Contexts	KT
Forest Contexts	KF

The data formulae describe the various parts of a DOM grove, whilst the context formulae describe contexts over the data structure that have a single hole which we can put data into.

3.2 Classical Formulae

The first set of formulae we shall look at are the classical logic formulae. We wish to be able to use standard results of first order logic, so we must be able to define this logic over our data and context structures. We will need to define the classical connectives over each formula type separately due to the multi-typed Grove, Tree and Forest data structure we are using. Note that we need only define the implies and false relations as the other first order logical relations, **true**, $\wedge$, $\vee$, $\neg$ and $\Leftrightarrow$, can be derived from these. Due to the multi-typed data

structure we are using, we will need a different false element for each of our formulae types. Whilst we list all six of these below, in practice we overload a single **false** element to represent false for all of the data formula types and a single **False** element to represent false for all of the context formula types, so we only refer to these overloaded elements in our satisfactions. From these we are able to define **true** for data and **True** for contexts in our logic.

Notation: We attach a type to each of our formula definitions so that it is always clear what kind of formula we are dealing with. This will be very important when we are dealing with context formula which can have different types depending on how the contexts are being used and what we are putting into our contexts. The formula definition $\mathbf{form} : \mathbf{T}$ states that the formula **form** is of type **T**.

Our logic provides the following classical formulae:

$$\begin{array}{rr} PG_1 \Rightarrow PG_2 : PG & KG_1 \Rightarrow KG_2 : KG \\ PT_1 \Rightarrow PT_2 : PT & KT_1 \Rightarrow KT_2 : KT \\ PF_1 \Rightarrow PF_2 : PF & KF_1 \Rightarrow KF_2 : KF \end{array}$$

$$\begin{array}{rr} \mathbf{false}_{PG} : PG & \mathbf{False}_{KG} : KG \\ \mathbf{false}_{PT} : PT & \mathbf{False}_{KT} : KT \\ \mathbf{false}_{PF} : PF & \mathbf{False}_{KF} : KF \end{array}$$

Notation: For the rest of this section we will be giving the forcing semantics for each of the different cases of formula satisfaction. To save space and needless repetition we choose to give the satisfactions in a parametric fashion. We will use variables **A** and **B** to represent any of G, the Grove formula, T, the Tree formula, or F, the Forest formula when the definitions are equivalent for each of the formula types. Note that $K\mathbf{A}$, $K\mathbf{B}$ are the context formula versions of these variables. The forcing semantics are then given using 2 satisfaction relations. The judgement $e, s, a \models_{DA} P\mathbf{A}$ states the the data formula $P\mathbf{A}$ holds for a given environment e, stack s and data a of type **A**, whilst judgement $e, s, ca \models_{CA} K\mathbf{A}$ states that the context formula $K\mathbf{A}$ holds for the environment e, the stack s and Context ca of type $C\mathbf{A}$.

Notation: The stack s is just as defined for the Operational semantics above and the environment e is a partial function over the DOM grove which sends Grove, Tree and Forest variables to their values.

$$e : (\mathbf{G} \to G) \times (\mathbf{T} \to T) \times (\mathbf{F} \to F) \times (\mathbf{CG} \to CG) \times (\mathbf{CT} \to CT) \times (\mathbf{CF} \to CF)$$

So we have the following obvious satisfactions for our classical formulae:

$$\begin{array}{rcl} e,s,a \models_{DA} P\mathbf{A}_1 \Rightarrow P\mathbf{A}_2 & \Leftrightarrow & e,s,a \models_{DA} P\mathbf{A}_1 \textit{ implies } e,s,a \models_{DA} P\mathbf{A}_2 \\ e,s,ca \models_{CA} K\mathbf{A}_1 \Rightarrow K\mathbf{A}_2 & \Leftrightarrow & e,s,ca \models_{CA} K\mathbf{A}_1 \textit{ implies } e,s,ca \models_{CA} K\mathbf{A}_2 \\ & e,s,a & \models_{DA} \textbf{false} \textit{ never} \\ & e,s,ca & \models_{CA} \textbf{False} \textit{ never} \end{array}$$

We also want to be able to talk about the equality of variable values, so we need to define the '=' operator which returns true if and only if the values on both side of the equality are the same. This is the only expression we are interested in, but we must be sure to define this over all four of our variable types: strings, pointers, integers and boolean values. We therefore provide the following expression formulae:

$$\begin{array}{ll} \mathbf{tag} = \mathbf{tag}' : PG & \mathbf{id} = \mathbf{id}' : PG \\ \mathbf{tag} = \mathbf{tag}' : PT & \mathbf{id} = \mathbf{id}' : PT \\ \mathbf{tag} = \mathbf{tag}' : PF & \mathbf{id} = \mathbf{id}' : PF \end{array}$$

$$\begin{array}{ll} \mathbf{n} = \mathbf{n}' : PG & \mathbf{bool} = \mathbf{bool}' : PG \\ \mathbf{n} = \mathbf{n}' : PT & \mathbf{bool} = \mathbf{bool}' : PT \\ \mathbf{n} = \mathbf{n}' : PF & \mathbf{bool} = \mathbf{bool}' : PF \end{array}$$

We are only interested in comparing data, not contexts, so it is therefore not necessary for us to define these expressions over context structures. Below we give the satisfactions for data formulae using the same parametric notation as set out above:

$$\begin{array}{rcl} e,s,a \models_{DA} \mathbf{tag} = \mathbf{tag}' & \Leftrightarrow & s(\mathbf{tag}) = s(\mathbf{tag}') \\ e,s,a \models_{DA} \mathbf{id} = \mathbf{id}' & \Leftrightarrow & s(\mathbf{id}) = s(\mathbf{id}') \\ e,s,a \models_{DA} \mathbf{n} = \mathbf{n}' & \Leftrightarrow & s(\mathbf{n}) = s(\mathbf{n}') \\ e,s,a \models_{DA} \mathbf{bool} = \mathbf{bool}' & \Leftrightarrow & s(\mathbf{bool}) = s(\mathbf{bool}') \end{array}$$

In order to be able to refer to the actual DOM grove, we need to define variables in our environment e. The reason for this is that DOM does not allow the storage of groves, trees or forests on the stack. We therefore need to use an environment whenever we wish to refer to an actual instance of a node in the grove. So we provide the following formulae for environments:

$$\begin{array}{ll} G : PG & CG : KG \\ T : PT & CT : KT \\ F : PF & CF : KF \end{array}$$

We give the satisfactions of these formulae using our parametric notation again:

$$\begin{array}{rcl} e,s,a \models_{DA} \mathbf{A} & \Leftrightarrow & a \equiv e(\mathbf{A}) \\ e,s,ca \models_{CA} C\mathbf{A} & \Leftrightarrow & ca \equiv e(C\mathbf{A}) \end{array}$$

Finally, for the classical formulae, we wish to define quantifiers for our variables. Note that we need only formally define $\exists$ as we can derive $\forall$ from existence since $\forall x.P = \neg\exists x.(\neg P)$. As before we must take care to define the quantifiers over all of our variable types, both for stack variables and for environment variables. Due to the large number of possible formula types we can construct using quantifiers, we shall make use of our parametric notation to define which formulae we can construct:

$$\begin{array}{rr}
\exists\,\mathbf{tag}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{tag}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{id}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{id}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{n}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{n}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{bool}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{bool}.K\mathbf{A} : K\mathbf{A}
\end{array}$$

$$\begin{array}{rr}
\exists\,\mathbf{G}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{G}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{T}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{T}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{F}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{F}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{CG}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{CG}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{CT}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{CT}.K\mathbf{A} : K\mathbf{A} \\
\exists\,\mathbf{CF}.P\mathbf{A} : P\mathbf{A} & \exists\,\mathbf{CF}.K\mathbf{A} : K\mathbf{A}
\end{array}$$

Sticking to our parametric notation we can show that we have the expected satisfactions:

$$\begin{array}{rcl}
e, s, a \models_{DA} \exists\,\mathbf{tag}.P\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{tag}.(e, s[\mathbf{tag} \mapsto \mathrm{tag}], a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{id}.P\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{id}.(e, s[\mathbf{id} \mapsto \mathrm{id}], a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{n}.P\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{n}.(e, s[\mathbf{n} \mapsto \mathrm{n}], a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{bool}.P\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{bool}.(e, s[\mathbf{bool} \mapsto \mathrm{bool}], a \models_{DA} P\mathbf{A})
\end{array}$$

$$\begin{array}{rcl}
e, s, ca \models_{CA} \exists\,\mathbf{tag}.K\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{tag}.(e, s[\mathbf{tag} \mapsto \mathrm{tag}], ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists\,\mathbf{id}.K\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{id}.(e, s[\mathbf{id} \mapsto \mathrm{id}], ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists\,\mathbf{n}.K\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{n}.(e, s[\mathbf{n} \mapsto \mathrm{n}], ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists\,\mathbf{bool}.K\mathbf{A} & \Leftrightarrow & \exists\,\mathrm{bool}.(e, s[\mathbf{bool} \mapsto \mathrm{bool}], ca \models_{CA} K\mathbf{A})
\end{array}$$

$$\begin{array}{rcl}
e, s, a \models_{DA} \exists\,\mathbf{G}.P\mathbf{A} & \Leftrightarrow & \exists\,G.(e[\mathbf{G} \mapsto G], s, a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{T}.P\mathbf{A} & \Leftrightarrow & \exists\,T.(e[\mathbf{T} \mapsto T], s, a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{F}.P\mathbf{A} & \Leftrightarrow & \exists\,F.(e[\mathbf{F} \mapsto F], s, a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{CG}.P\mathbf{A} & \Leftrightarrow & \exists\,CG.(e[\mathbf{CG} \mapsto CG], s, a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{CT}.P\mathbf{A} & \Leftrightarrow & \exists\,CT.(e[\mathbf{CT} \mapsto CT], s, a \models_{DA} P\mathbf{A}) \\
e, s, a \models_{DA} \exists\,\mathbf{CF}.P\mathbf{A} & \Leftrightarrow & \exists\,CF.(e[\mathbf{CF} \mapsto CF], s, a \models_{DA} P\mathbf{A})
\end{array}$$

$$
\begin{array}{rcl}
e, s, ca \models_{CA} \exists \mathbf{G}.K\mathbf{A} & \Leftrightarrow & \exists G.(e[\mathbf{G} \mapsto G], s, ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists \mathbf{T}.K\mathbf{A} & \Leftrightarrow & \exists T.(e[\mathbf{T} \mapsto T], s, ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists \mathbf{F}.K\mathbf{A} & \Leftrightarrow & \exists F.(e[\mathbf{F} \mapsto F], s, ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists \mathbf{CG}.K\mathbf{A} & \Leftrightarrow & \exists CG.(e[\mathbf{CG} \mapsto CG], s, ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists \mathbf{CT}.K\mathbf{A} & \Leftrightarrow & \exists CT.(e[\mathbf{CT} \mapsto CT], s, ca \models_{CA} K\mathbf{A}) \\
e, s, ca \models_{CA} \exists \mathbf{CF}.K\mathbf{A} & \Leftrightarrow & \exists CF.(e[\mathbf{CF} \mapsto CF], s, ca \models_{CA} K\mathbf{A})
\end{array}
$$

So far we have not defined anything that is new or surprising. This style of logic is well defined and has been used many times before. However, what we go on to define next is new and specific to our needs.

3.3 Structural Formulae

We turn our attention to the structure of the data and contexts provided by our Grove, Tree and Forest approach to viewing the DOM data structure. First we need to be able to describe the layout of any grove that we can construct from our data structure definition. We give the following formulae:

$$
\begin{array}{rr}
\emptyset : PG & \_ : KG \\
PG_1 \oplus PG_2 : PG & KG \oplus PG : KG \\
PT : PG & KT : KG \\
 & \\
\mathbf{tag_{id}}[PF]_{\mathbf{fid}} : PT & \mathbf{tag_{id}}[KF]_{\mathbf{fid}} : KT \\
 & \\
PF ::= \emptyset : PF & \_ : KF \\
PF_1 \times PF_2 : PF & KF \times PF : KF \\
PT : PF & PF \times KF : KF \\
 & KT : KF
\end{array}
$$

We have an exact correspondence here between the above formulae and the data structure definitions given in section 1.5.3. For example $PG_1 \oplus PG_2$ describes a grove which we can split using the $\oplus$ operator to give a grove that satisfies PG_1 and a grove which satisfies PG_2. Similarly $\mathbf{tag_{id}}[PF]_{\mathbf{fid}}$ describes a node with a tag, identifier and forest identifier as given and with a forest of children which satisfies PF.

We provide the satisfactions of these formulae in the following fashion:

$$
\begin{array}{rcl}
e,s,G \models_{DA} \emptyset & \Leftrightarrow & G \equiv \emptyset \\
e,s,G \models_{DA} PG_1 \oplus PG_2 & \Leftrightarrow & \exists G_1, G_2.(e,s,G_1 \models_{DA} PG_1 \; and \; e,s,G_2 \models_{DA} PG_2 \; and \; G \equiv G_1 \oplus G_2) \\
e,s,G \models_{DA} PT & \Leftrightarrow & \exists T.(e,s,T \models_{DA} PT \; and \; G \equiv T) \\
\\
e,s,T \models_{DA} \mathbf{tag_{id}}[PF]_{\mathbf{fid}} & \Leftrightarrow & \exists F.(e,s,F \models_{DA} PF \; and \; T \equiv s(\mathbf{tag})_{s(\mathbf{id})}[F]_{s(\mathbf{fid})}) \\
\\
e,s,F \models_{DA} \emptyset & \Leftrightarrow & F \equiv \emptyset \\
e,s,F \models_{DA} PF_1 \times PF_2 & \Leftrightarrow & \exists F_1, F_2.(e,s,F_1 \models_{DA} PF_1 \; and \; e,s,F_2 \models_{DA} PF_2 \; and \; F \equiv F_1 \times F_2) \\
e,s,F \models_{DA} PT & \Leftrightarrow & \exists T.(e,s,T \models_{DA} PT \; and \; F \equiv T) \\
\\
e,s,CG \models_{CA} \_ & \Leftrightarrow & CG \equiv \_ \\
e,s,CG \models_{CA} KG \oplus PG & \Leftrightarrow & \exists CG', G.(e,s,CG' \models_{CA} KG \; and \; e,s,G \models_{DA} PG \; and \; CG \equiv CG' \oplus G) \\
e,s,CG \models_{CA} KT & \Leftrightarrow & \exists CT.(e,s,CT \models_{CA} KT \; and \; CG \equiv CT) \\
\\
e,s,CT \models_{CA} \mathbf{tag_{id}}[KF]_{\mathbf{fid}} & \Leftrightarrow & \exists CF.(e,s,CF \models_{CA} KF \; and \; CT \equiv s(\mathbf{tag})_{s(\mathbf{id})}[CF]_{s(\mathbf{fid})}) \\
\\
e,s,CF \models_{CA} \_ & \Leftrightarrow & CF \equiv \_ \\
e,s,CF \models_{CA} KF \times PF & \Leftrightarrow & \exists CF', F.(e,s,CF' \models_{CA} KF \; and \; e,s,F \models_{DA} PF \; and \; CF \equiv CF' \times F) \\
e,s,CF \models_{CA} PF \times KF & \Leftrightarrow & \exists F, CF'.(e,s,F \models_{DA} PF \; and \; e,s,CF' \models_{CA} KF \; and \; CF \equiv F \times CF') \\
e,s,CF \models_{CA} KT & \Leftrightarrow & \exists CT.(e,s,CT \models_{CA} KT \; and \; CF \equiv CT)
\end{array}
$$

What remains is for us to logically define the use of contexts. We require formulae which specify how we can apply the different contexts to the different data and context types. There are 3 types of context formulae. The simplest of these is context application $K\mathbf{A}(P\mathbf{B})$, which describes a grove which we can split into a context satisfying $K\mathbf{A}$ and some data satisfying $P\mathbf{B}$. This results in a completed context of type $P\mathbf{A}$. We have the formulae definitions:

$$
\begin{array}{lll}
KG(PG) : PG & KT(PT) : PT & KF(PT) : PF \\
KG(PT) : PG & KT(PF) : PT & KF(KF) : PF \\
KG(PF) : PG &
\end{array}
$$

Notice that we do not allow the insertion of a grove into a tree or forest context. This is simply because these cases cannot occur. Recall that the top level of DOM is an unordered set of DOM trees and that all nodes beneath this contain forests that are ordered lists of trees. It does not make sense to put an unordered set of trees into an ordered list, so we do not allow this context construction. In effect, it is this omission which allows us to reason over the

DOM tree structure without having to carry around a notion of tree depth. Since the context application for each context type follows the same simple structure we once again make use of our parametric notation to give the satisfaction,

$$e, s, a \models_{DA} K\mathbf{A}(P\mathbf{B}) \quad \Leftrightarrow \quad \exists ca \in C\mathbf{A}, b \in P\mathbf{B}.(\mathrm{ap}(ca, b) = a \textit{ and } e, s, ca \models_{CA} K\mathbf{A} \textit{ and } e, s, a \models_{DA} P\mathbf{B})$$

The next type of context formulae are the $\triangleright$ or 'right triangle' formulae.
$P\mathbf{B} \triangleright P\mathbf{A}$ describes a context where, whenever we put data satisfying $P\mathbf{B}$ into the context, the resulting completed context satisfies $P\mathbf{A}$. This is context logic's adjoint to separation logic's $-\!\!*$ operator [2]. These formulae are of type $K\mathbf{A}$ and we give the following formulae definitions:

$$\begin{array}{ccc} PG_1 \triangleright PG_2 : KG & PT_1 \triangleright PT_2 : KT & PT \triangleright PF : KF \\ PT \triangleright PG : KG & PF \triangleright PT : KT & PF_1 \triangleright PF_2 : KF \\ PF \triangleright PG : KG & & \end{array}$$

As in the previous case we do not allow the construction of any formulae which would describe a tree or forest context where a grove can be place inside it. We can give the satisfaction for formulae of this kind thusly,

$$e, s, ca \models_{CA} P\mathbf{B} \triangleright P\mathbf{A} \quad \Leftrightarrow \quad \forall b \in P\mathbf{B}.(e, s, b \models_{DA} P\mathbf{B} \textit{ implies } e, s, \mathrm{ap}(ca, b) \models_{DA} P\mathbf{A})$$

The final type of context formulae are the $\triangleleft$ or 'left triangle' formulae.
$K\mathbf{A} \triangleleft P\mathbf{A}$ describes a piece of data where, whenever we warp a context satisfying $K\mathbf{A}$ around the data, we get a completed context that satisfies $P\mathbf{A}$. These formulae can be of any type $P\mathbf{B}$ except that we must remember that we cannot put a grove into a tree or forest context. Keeping this in mind we can define the following formulae:

$$\begin{array}{ccc} KG \triangleleft PG : PG & KT \triangleleft PT : PT & KF \triangleleft PF : PT \\ KG \triangleleft PG : PT & KT \triangleleft PT : PF & KF \triangleleft PF : PF \\ KG \triangleleft PG : PF & & \end{array}$$

We give the satisfaction for these formulae as,

$$e, s, b \models_{DA} K\mathbf{A} \triangleleft P\mathbf{A} \quad \Leftrightarrow \quad \forall ca \in C\mathbf{A}.(e, s, ca \models_{CA} K\mathbf{A} \textit{ implies } e, s, \mathrm{ap}(ca, b) \models_{DA} P\mathbf{A})$$

We now have a logic that is just as precise as our operational semantics, which is expressible enough to describe our new data structure and, as we will show in the next sections, is capable of being used in a compositional manor that allows us to very easily build up a complete specification for DOM Core Level 1 from Minimal DOM.

3.4 Logic Examples

Before we get on to using this logic, let us first consider some examples to illustrate how we will be using this logic in the following sections of this report. We will start with some very simple examples to show the basic ways in which we use this logic and we shall go on to give a few, more complex, examples of formula styles we will be using throughout the remaining sections of this report.

Example 1 - Small Tree

To start we will consider the very simple tree, consisting of just 3 nodes, as given in figure 11.

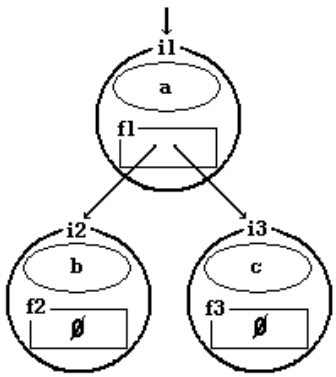

Figure 11: Small Tree

We would represent figure 11 by the formula:

$$\mathrm{a}_{i1}[\mathrm{b}_{i2}[\emptyset]_{f2} \times \mathrm{c}_{i3}[\emptyset]_{f3}]_{f1}$$

This is exactly the same style as we would get using our data structure definitions, but we now have a logical formula expressing the same information. Only a grove with the exact structure given in figure 11 can satisfy this formula.

Example 2 - Top Node

We now move onto a more interesting example, where instead of describing exactly what our tree looks like we give a property we wish our tree to possess. Referring to Figure 11 again we now wish to give a formula that specifies that our tree has Node $i1$ at its top level. To do this we make use of the fact that the expression '**true**' is always satisfied. So we express this property as,

$$\mathrm{a}_{i1}[\mathbf{true}]_{f1}$$

Any grove which has the Node $\mathrm{a}_{i1}[\mathbf{F}]_{f1}$ at its top level can satisfy this formula, figure 11 is just one possible case. The use of **true** means that the forest **F** can be anything, from the empty forest to a wide and deep subtree.

Example 3 - Middle Node

Similarly we can use the formula,

$$\exists\,(\mathbf{tag}, \mathbf{id}, \mathbf{fid}).\mathbf{tag_{id}}[\mathbf{true} \times \mathrm{c}_{i3}[\mathbf{true}]_{f3}]_{\mathbf{fid}}$$

to describe a grove where the node $i3$ is at the second level to the far right of the list of children it appears in. Again figure 11 is just one possible grove which satisfies this formula.

Example 4 - Node Existence

One of the key properties that we will want to be able to express about groves is that a specific Node exists in the grove. All of our commands require the existence of the Node(s) they are called to operate upon, so we need to be able to describe groves which contain specific Nodes. To do this for the general case we need to make use of context formulae. One way that we are able to define such a grove is as follows,

$$\mathbf{True}(\mathrm{c}_{i3}[\mathbf{true}]_{f3})$$

This says that the Node $i3$ occurs somewhere in the grove, specifically we have some arbitrary context which has the Node $i3$ in its hole with an arbitrary subtree beneath the Node $i3$. Whilst this formula correctly describes the existence property we desire, it makes no constraints upon the rest of the grove.

Example 5 - Context Variables

The style of formula presented above in Example 4 is not constraining enough for our purposes. We could, for example, give the pre-condition for some get command that does not modify the grove in the above form. Naively we might assume that the post-condition would be the same formula, plus the description of the information we have retrieved. This, however, is not the case. The **True** formula imposes no restriction on the data that satisfies it, specifically **True** in the pre-condition could be satisfied by a different grove context to the **True** in the post-condition. Likewise for **true**. So we are not actually guaranteeing that the grove is unchanged by the command at all. In order to correctly state that the grove has not changed by the command, we instead require a formula of the form,

$$\mathbf{CG}(\mathrm{c}_{i3}[\mathbf{F}]_{f3})$$

Here we make use of a context variable **CG** and a forest variable **F** which have values assigned to them in the environment e. The use of such variables in the pre- and post-conditions does indeed specify that the command does not modify the grove so long as the environment e is unchanged. Note that since we do not put any constraints on the values of these variables, we still have a formula which is satisfied by any grove that contains the Node $i3$ somewhere in it. This formula makes no assumptions about the depth of Node $i3$, so it could appear at the top level or at any level deeper than this.

Example 6 - Not An Ancestor

You may recall that in section 1 we mentioned that the move commands of Minimal DOM will only execute correctly if the child to be move is not an ancestor of the parent it is to be moved beneath. We will now show how our logic allows us to neatly express this property of a grove. As in the previous example, it is the use of context formulae that allows us to describe this property in a formal fashion.
We begin by considering the grove context formula,

$$(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}))$$

Remember that the $\mathbf{P} \triangleright \mathbf{Q}$ operator defines a context that satisfies the formula **Q** whenever we put into it data satisfying the formula **P**. So the above context satisfies $\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}})$ whenever we put the empty data (i.e. no data) into the context hole. As we just explained in Example 5, this describes the grove that contains the Node **parent** somewhere within it. In effect, what we are really saying here is that the context hole cannot be above the Node **parent**. By our construction of contexts we do not allow anything to occur beneath a context hole, so all context holes must occur as a leaf of a tree (i.e. they have no children). The context we are describing must have both a context hole and the Node **parent** somewhere within itself. Now if we place some other data into the context hole instead of $\emptyset$, say the child we wish to move, then this data still cannot be an ancestor of the Node **parent** because we have not changed the structure of the context, just the structure of the data. The formula,

$$(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}))(\mathbf{tag}'_{\mathbf{child}}[\mathbf{F}']_{\mathbf{fid}'})$$

specifies just that. This can only be satisfied by a grove which contains both the **parent** and **child** Nodes and, moreover, must have the child in the tree so that it is not an ancestor of the parent. Note that the use of context variable **CG** and forest variables **F** and $\mathbf{F}'$ allows for any arbitrary grove to be built around the **parent** and **child** Nodes. This formula would also be suitable for use as a pre- or post-condition for a command which does not modify the parts of the grove contained in the **CG**, **F** or $\mathbf{F}'$.

Example 7 - Weakest Pre-condition

In section 5 we will properly introduce the concept of a weakest pre-condition, along with formally deriving the weakest pre-conditions for each of the commands of Minimal DOM, but as a final example of the logic we have defined for our data structure let us look at the weakest pre-condition for the append command.

$$(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}) \triangleright P)((\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathfrak{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}))$$

The post-condition for the command is P, so the above pre-condition is the weakest, or least constraining, pre-condition that will guarantee P after the append command has executed. To understand what this pre-condition is describing, let us break it down into smaller pieces. First note that the last half of this pre-condition,

$$((\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathfrak{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}))$$

is exactly the formula we considered in Example 6 above. This grove data formula describes a grove which contains both some Node **parent** and some Node **newChild** where **newChild** is not an ancestor of **parent**. Now this grove is contained inside a context which is described by the first half of the formula. The context formula,

$$(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}) \triangleright P)$$

states that if the data it is wrapped around were to satisfy

$$\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}})$$

then the resulting grove would satisfy P. But this data formula simply states that the **newChild** Node is the last child of the **parent** Node. So what the entire pre-condition says is - currently we have a grove which contains the Nodes **newChild** and **parent** where **newChild** is not an ancestor of **parent**, but if **newChild** were to be moved to be the last child of **parent** then the grove would now satisfy the data formula P. Since the append command takes such a Node **newChild** and moves it to the end of the list of children for **parent**, after the command has executed we will indeed have a grove which satisfies the data formula P. Notice that the careful use of grove and forest variables **CG**, **F** and $\mathbf{F}'$ describes that the rest of the grove is not modified by the append command, i.e. the command behaves in a local fashion.

4 Hoare Reasoning

We mentioned previously that we wanted to create a specification in a compositional manor. To create this compositional power we turn to the use of Hoare reasoning. Hoare reasoning is concerned with the definition and derivation of Hoare Triples over a set of commands. The first step is to reason about the local behaviour of our commands, that is their behaviour over the parts of the grove that are modified by their execution. We then use the general Hoare reasoning framework to take our reasoning to the complete data structure and produce a full specification of our commands.

4.1 The Reasoning Framework

First we give the reasoning framework for Hoare logic. We start by looking at Local Hoare Triples, which are of the form,

{pre-condition}
Command
{post condition}

As usual, the pre-condition specifies the state of the data structure before the command has been called and the post-condition specifies the state of the data structure after the command has been fully executed. However a Local Hoare Triple has a stronger meaning than this alone. For any Grove G and stack s, then for any of our commands $\mathbb{C}$ we have the Triple,

$$\{P\}\mathbb{C}\{Q\} \Leftrightarrow s, G \models P \Rightarrow (\mathbb{C}, s, G \not\rightsquigarrow \text{fault}) \; and \; \exists\, s', G' \models Q.(\mathbb{C}, s, G \rightsquigarrow s', G')$$

In other words, for any Grove and stack that satisfy the pre-conditions of the commands, the command will not fail and moreover the resulting Grove and stack after the command will satisfy the post-condition.

Simple Example

For a toy example of how these commands are intended to be read let us consider a state capable of storing integers and the add1(**var**) command which increments the value stored at **var** by 1. This command would have the following Local Hoare Triple associated with it:

$$\begin{array}{l}\{x = n\}\\ \text{add1}(x)\\ \{x = n + 1\}\end{array}$$

In the obvious fashion, $x = n$ specifies a state which has the value n stored in variable x.

The above command is in fact what we call the "Small Axiom" for the add1 command. A command's Small Axiom is the associated Local Hoare Triple with the smallest possible footprint for that command. Recall that the footprint of a command is the part of the data structure that the command needs to know about if it is to run correctly and also any part of the data structure that is modified by the command. In the above example the smallest footprint of the add1 command is just the value of the variable **var** itself. We will see later that from these axioms it is easy to define the behavior of the command on the overall data structure.

Once we have the Local Hoare Triples for each command, we can see how multiple commands will affect the data structure. This is a simple task for Hoare reasoning as we can build up the more complex commands from a number of smaller commands and obtain the behaviour of the new command with little extra reasoning. To show what we mean by this let us consider an extension of the toy example given above.

Example: Using Local Hoare Triples:

Suppose we want to specify the Local Hoare Triple of the command add2(**var**) which behaves in exactly the same manor as add1(**var**), except that it adds 2 to the variable **var** instead of 1. Rather than defining the Local Hoare Triple directly, we could instead note that add2(**var**) $\equiv$ {add1(**var**); add1(**var**)}. This notion can be carried through the Local Hoare Triples of the command as follows:

$$\begin{aligned}&\{x = n\}\\&\text{add1}(x)\\&\{x = n + 1\}\\&\text{add1}(x)\\&\{x = n + 1 + 1\}\\&\{x = n + 2\}\end{aligned}$$

This composition gives us the following Small Axiom for the add2 command:

$$\begin{aligned}&\{x = n\}\\&\text{add2}(x)\\&\{x = n + 2\}\end{aligned}$$

Now with such a trivial example as given above we haven't really saved ourself any work, but with bigger and more complex commands the utility of this

procedure is invaluable. We will demonstrate the power of this technique later when we show how to derive the specification for the remainder of structural DOM Core level 1, and more, from Minimal DOM. In effect we are looking to specify the local behavior of the building blocks for Minimal DOM that will then allow us to define the entire structural behaviour of DOM Core level 1 via the general context logic framework.

4.1.1 Hoare Rules

Although Context Logic is still a relatively new subject area, it already has a well established framework for dealing with the move from local reasoning to higher level reasoning. We shall now discuss the rules which will be central to the movement from our local Axioms to the more general Weakest Pre-conditions. Hoare Reasoning has several general inference rules for Local Hoare Triples which are detailed below. In the statement of these rules we assume sets of free variables free($\mathbb{C}$) and modified variables mod($\mathbb{C}$)of a command $\mathbb{C}$. Intuitively the set free($\mathbb{C}$) is the set of variables that may affect the execution of command $\mathbb{C}$ and mod($\mathbb{C}$) is the set of variables that the command $\mathbb{C}$ may modify. We express these properties formally as follows:

$$
\begin{array}{ll}
\text{if} & x \notin \text{free}(\mathbb{C}) \text{ then,} \\
& \mathbb{C}, s, G \rightsquigarrow \text{fault} \Rightarrow \mathbb{C}, s[x \mapsto v], G \rightsquigarrow \text{fault} \\
& \mathbb{C}, s, G \rightsquigarrow s', G' \Rightarrow \mathbb{C}, s[x \mapsto v], G \rightsquigarrow s'[x \mapsto v], G' \\
\text{if} & x \notin \text{mod}(\mathbb{C}) \text{ then,} \\
& \mathbb{C}, s, G \rightsquigarrow s', G' \Rightarrow s(x) = s'(x)
\end{array}
$$

Our rules require that these properties of free($\mathbb{C}$) and mod($\mathbb{C}$) be satisfied for them to be sound.

Consequence Rule:

The Consequence rule gives us the ability to derive pre- and post- conditions from one another. $P' \Rightarrow P$ is classical consequence which means that P' logically entails P. The rule is given as,

$$
\frac{P' \Rightarrow P \quad \{P\}\mathbb{C}\{Q\} \quad Q \Rightarrow Q'}{\{P'\}\mathbb{C}\{Q'\}}
$$

which allows us to weaken the pre-condition and tighten the post-condition. This is very important for our ability to derive the weakest pre-conditions for our commands from their axioms.

Auxiliary Variable Elimination Rule:

Variable elimination simply allows us to quantify over our variables so that our reasoning can be made applicable to the general case.

$$\frac{\{P\}\mathbb{C}\{Q\}}{\{\exists x.P\}\mathbb{C}\{\exists x.Q\}} \quad {\scriptstyle x \notin \text{free}(\mathbb{C})}$$

Sequencing Rule:

Sequencing provides us with the compositional power of Local Hoare Reasoning, allowing us to bolt together the axioms for our commands so that we may derive the axioms for larger, more complex, commands with little extra reasoning required.

$$\frac{\{P\}\mathbb{C}_1\{Q\} \quad \{Q\}\mathbb{C}_2\{R\}}{\{P\}\mathbb{C}_1;\mathbb{C}_2\{R\}}$$

The only work that needs to be done is using Consequence to enable us to make the post-condition of one command be in the correct form for the pre-condition for the next command.

Frame Rule:

The final, and most important, of our rules is the Frame Rule. This rule gives us the ability to take our reasoning from the local, axiom, level up to the full DOM grove by the application of contexts.

$$\frac{\{P\}\mathbb{C}\{Q\}}{\{K(P)\}\mathbb{C}\{K(Q)\}} \quad {\scriptstyle \text{mod}(\mathbb{C}) \cap \text{free}(K) = \emptyset}$$

4.1.2 Local Commands

Whilst the first three of these rules are standard Hoare logic rules, the use of the Frame Rule relies on the fact that all of our commands behave locally. That is that they satisfy both the "Safety-Monotonicity Property" and the "Frame Property".

Safety-Monotonicity Property

For any Grove G, any Grove Context CG and any of our commands $\mathbb{C}$, we have,

$$\mathbb{C}, s, G \not\rightsquigarrow \text{fault} \wedge \text{ap}(CG, G)\downarrow \Rightarrow \mathbb{C}, s, \text{ap}(CG, G) \not\rightsquigarrow \text{fault}$$

This simply states that if our commands do not fault on some data and we have a defined application of some context to that data, then the commands will

still not fault when run in that context on the same data. In other words, our commands will still work on some data, even if we wrap a context around that data, so long as that context application is defined.

Frame Property

For any Groves G and G', any Grove Context CG, any stacks s and s' and any of our commands $\mathbb{C}$ we have,

$$\mathbb{C}, s, G \not\rightsquigarrow \text{fault} \wedge \text{ap}(CG,\, G) \downarrow \wedge\, \mathbb{C}, s, \text{ap}(CG,\, G) \rightsquigarrow s', G' \Rightarrow \exists G''.(\mathbb{C}, s, G \rightsquigarrow s', G'' \wedge G' = \text{ap}(CG,\, G''))$$

This states that if our commands do not fault on some data, we have a defined application of some context to that data and the command run in this context goes to some program state, then there exists some third data which is the result of the command run on our original data and applying our context to this third data type gives the same result as the command run in the context applied to the original data type. In other words, if we apply a context to some data and run the command, we get the same end data as if we run the command on some data and then apply the context to the data output of the command, so long as the context application is defined.

It has been shown [9] that commands specified by Local Hoare Triples do in fact satisfy both of these properties, so all of the above rules are sound.

4.2 Formulating DOM

Now that we have a reasoning framework in place we can move on to formulating a full specification for Minimal DOM. We already have a logic capable of of expressing program states for Minimal DOM, so using this logic we can provide pre- and post-conditions for our commands. As with the example given above we wish to specify just the local behaviour of each command. We give pre- and post-conditions over the smallest footprints of each command. These command axioms will then enable us to derive the full weakest- pre-conditions for each command which will provide the full specification for Minimal DOM. To illustrate the meaning of our axioms we will first take the append command as an example.

4.2.1 Append Axiom

The axiom for our append command can be given as,

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}'})\}$
append(**parent**, **newChild**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \text{ x } \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}'}]_{\mathbf{fid}})\}$

The pre- and post-conditions here share a lot in common with Examples 6 and 7 of the previous section. The pre-condition specifies a grove where both of the Nodes **parent** and **newChild** exist and where **newChild** is not an ancestor or **parent**. The postcondition specifies a grove where these two Nodes still exist, but now **newChild** is the last child of **parent**.

4.2.2 Axioms

The Axioms for all of the commands of Minimal DOM are given below,

$\{\emptyset\}$
$\mathbf{node}'$ = createNode(**tag**);
$\{\emptyset \oplus \mathbf{tag}_{\mathbf{node}'}[\emptyset]_{\mathbf{fid}}\}$

$\{\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{tag}'$ = getNodeName(**node**);
$\{\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{tag}' = \mathbf{tag}\}$

$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}\}$
forest = getChildNodes(**parent**);
$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{forest} = \mathbf{fid}\}$

$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \text{x } \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \text{ x } \mathbf{F}_2]_{\mathbf{fid}}\}$
$\mathbf{parent}'$ = getParentNode(**child**);
$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \text{x } \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \text{ x } \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathbf{parent}' = \mathbf{parent}\}$

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \text{x } \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathbf{fid}'} \text{ x } \mathbf{F}_2]_{\mathbf{fid}}))(\mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}''})\}$
insertBefore(**parent**, **newChild**, **ref**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \text{x } \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}''} \text{ x } \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathbf{fid}'} \text{ x } \mathbf{F}_2]_{\mathbf{fid}})\}$

$\{\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \text{x } \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \text{ x } \mathbf{F}_2]_{\mathbf{fid}})\}$
node = removeChild(**parent**, **child**);
$\{\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \text{x } \mathbf{F}_2]_{\mathbf{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \wedge \mathbf{node} = \mathbf{child}\}$

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}'})\}$
append(**parent**, **newChild**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \text{x } \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}'}]_{\mathbf{fid}})\}$

To assist with the definition of the Small Axioms for the Nodelist structure, we add the the lng predicate, for list length, to our logic. This predicate extends the logic of integers in the obvious fashion enabling us to check the equality of the predicate with other integer variables or values. The definition of the lng predicate is given below:

$$\begin{gathered}\mathrm{lng}(\emptyset) = 0 \\ \mathrm{lng}(\mathbf{T} \times \mathbf{F}) = 1 + \mathrm{lng}(\mathbf{F})\end{gathered}$$

Recall that we have assumed our lists to be 0 indexed.

$\{\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{length} = \mathrm{getLength}(\mathbf{fid});$
$\{\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{length} = \mathrm{lng}(\mathbf{F})\}$

$\{\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \ \mathrm{lng}(\mathbf{F}_1) = \mathbf{length}\}$
$\mathbf{node} = \mathrm{getItem}(\mathbf{length}, \mathbf{fid});$
$\{\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \wedge \mathbf{node} = \mathbf{item}\}$

4.2.3 Use of Notation Overload

You will recall that earlier we mentioned how the $\emptyset$ element is overloaded in our notation to express both the empty Grove (or set) and the empty Forest (or list). We have made use of this notation overload above to provide simpler axioms. Consider the commands with preconditions of the form $\emptyset \triangleright P$. The notational overload means that this single pre-condition expresses both the cases $\emptyset_g \triangleright P$ and $\emptyset_f \triangleright P$, where $\emptyset_g$ and $\emptyset_f$ represent the empty set and empty list respectively. So rather than needing a complex condition on these contexts splitting apart the cases where the context hole appears at the top level or within some subtree of the grove, we use our notational overload to express both of these cases in one neat formula.

4.2.4 Axiom Evaluation

The main point of interest that should be mentioned here is that the above axioms are not all strictly small in the normal sense. Specifically the axioms provided for the move commands "insertBefore", "removeChild" and "append" can potentially have very large footprints due to the context variables used to define them. To explain the implications of this further let us take the "append" command and look at its axiom in more detail.

Append - A Closer Look:

As we discussed earlier, in order to both specify that the Node **newChild** is not an ancestor of the Node **parent** and that the append command does not modify any other part of the grove other than the position of the **newChild**

Node, we have to make use of context and forest variables. However, the very general nature of these variables, which allows us to specify that the rest of the grove is unmodified, also causes the footprint of this command to be of an arbitrary size. The smallest footprint would be the subtree of the grove which contains both **newChild** and **parent** with as little of the rest of the grove as possible. Unfortunately, the use of a context variable **CG** only requires that the grove be big enough to contain both **newChild** and **parent**. Specifically this variable allows the footprint to be anything from the smallest footprint to the entire DOM grove.

The potentially large nature of these footprints does not stop the Local Hoare Triples from holding, it just means that the axioms are not truly small in the three move cases. Some work has been undertaken at providing strengthened Local Hoare Triples that include an assertion that the contexts for these commands must be as small as possible, but we have yet to produce satisfactory results that maintain our neat notation style. However, it is agreed that these "Medium Axioms" are still sufficient for the purpose of specifying the behaviour of the move commands and as such we will put any concerns about the footprint size of these commands to one side at this time.

4.2.5 Soundness Theorem:

The small axioms are sound.

proof: This result follows directly from the operational semantics of the commands.

5 Deriving the Weakest Pre-conditions

In [9] there is a detailed discussion of how the weakest pre-condition for a command can be derived from its Small Axiom. Making use of this style of reasoning we will now show the derivations for the weakest pre-conditions of each command in Minimal DOM. Obtaining the Weakest Pre-conditions from the axioms of our commands guaranties the safety of the commands. This means they will not fault so long as the data structure satisfies the pre-conditions of the Local Hoare Triples.

The proofs in this section make use of the fact that our data structure is precise. This gives us the following property:

$$(Q \triangleright P)(Q) \Leftrightarrow \mathrm{True}(Q) \wedge P$$

The reverse part of this property $\Leftarrow$, is not true in general, but it is true in the case of a precise data structure.

5.1 Notation and Proof Structure

Throughout this section we shall be using the reasoning framework set out in the previous section. Specifically there shall be a reference to which Hoare Rule was applied at each proof step in order to derive the next pre-, post-condition pair:

cons - refers to the consequence rule

aux - refers to the auxiliary variable elimination rule

seq - refers to the sequencing rule

frame - refers to the frame rule

The derivation for each command will follow a series of proof steps, starting from the axioms as set out in the previous section, and ending with the weakest pre-condition for that command. Each proof step will be separated by a horizontal line and what is above the line will directly entail what is beneath the line using the Hoare Rule noted. So for example,

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{K(P)\}\ \mathbb{C}\ \{K(Q)\}} \quad (\textbf{frame})$$

states that we use the **Frame Rule** to wrap context K around the pre-condition P and the post-condition Q.

5.2 append - a Detailed Proof

Before we give the full set of derivations, let us first take the append command and give each of the proof steps in full detail. This detailed derivation should serve as a guideline for the reasoning in the remaining cases.

We begin with the axiom for the append command,

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathfrak{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'})\}$
append(**parent**, **newChild**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \mathbf{x}\ \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}})\}$ }

Now in order for us to be able to derive a post-condition P, we need to ensure that once the append command has executed, that P is satisfied by a grove where **newChild** is the last child under **parent**. Now if you recall, from the examples of our logic, we were able to define a context that described just that. Namely the formula,

$$K = (\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}) \triangleright P)$$

So using the **Frame Rule**, we can apply this context to the grove we have already described in the axiom of the command. This gives us the new pre-, post-condition pair

$\{(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}) \triangleright P)((\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathfrak{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}))\}$
append(**parent**, **newChild**);
$\{(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}) \triangleright P)(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}))\}$

Notice that we apply the context K to both the pre- and post-condition of the command. The next step makes use of the **Consequence Rule**. Notice that in the post-condition the data within the context is,

$$(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}})) \tag{1}$$

Which means that the post-condition is in the form $(Q \triangleright P)(Q)$ and this $\Rightarrow$ True$(Q) \wedge P \Rightarrow P$ as we mentioned above. Since we always have that $F \Rightarrow F$, for any formula F, we do not need to change the pre-condition. So we apply **cons** to get,

$\{(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F} \times \mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}]_{\mathfrak{fid}}) \triangleright P)((\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathfrak{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'}))\}$
append(**parent**, **newChild**);
$\{P\}$

We now have the post-condition P, so the pre-condition is the weakest it can be and still guarantee P, i.e. it is the weakest pre-condition for the command. This completes the derivation for the append command.

5.3 The Derivations

We now give the derivations for the remaining commands of Minimal DOM. Recall that each proof begins with the command's axiom and that each step uses one of our Hoare Rules as noted. In each case the pre-condition of the final step, where the post-condition is just P, is the weakest pre-condition for that command.

5.3.1 createNode

$\{\emptyset\}$
$\mathbf{node}' = \text{createNode}(\mathbf{tag});$
$\underline{\{\emptyset \oplus \mathbf{tag}_{\mathbf{node}'}[\emptyset]_{\mathtt{fid}}\}}$ (**frame**)
$\{(\forall \mathbf{node}, \mathbf{fid}(\mathbf{tag}_{\mathbf{node}}[\emptyset]_{\mathtt{fid}}) \triangleright P[\mathbf{node}/\mathbf{node}'])(\emptyset)\}$
$\mathbf{node}' = \text{createNode}(\mathbf{tag});$
$\underline{\{(\forall \mathbf{node}, \mathbf{fid}(\mathbf{tag}_{\mathbf{node}}[\emptyset]_{\mathtt{fid}}) \triangleright P[\mathbf{node}/\mathbf{node}'])(\emptyset \oplus \mathbf{tag}_{\mathbf{node}'}[\emptyset]_{\mathtt{fid}}\}}$ (**cons**)
$\{(\forall \mathbf{node}, \mathbf{fid}(\mathbf{tag}_{\mathbf{node}}[\emptyset]_{\mathtt{fid}}) \triangleright P[\mathbf{node}/\mathbf{node}'])(\emptyset)\}$
$\mathbf{node}' = \text{createNode}(\mathbf{tag});$
$\underline{\{(\forall \mathbf{node}, \mathbf{fid}(\mathbf{tag}_{\mathbf{node}}[\emptyset]_{\mathtt{fid}}) \triangleright P[\mathbf{node}/\mathbf{node}'])(\mathbf{tag}_{\mathbf{node}'}[\emptyset]_{\mathtt{fid}})\}}$ (**cons**)
$\{(\forall \mathbf{node}, \mathbf{fid}(\mathbf{tag}_{\mathbf{node}}[\emptyset]_{\mathtt{fid}}) \triangleright P[\mathbf{node}/\mathbf{node}'])(\emptyset)\}$
$\mathbf{node}' = \text{createNode}(\mathbf{tag});$
$\underline{\{P[\mathbf{node}/\mathbf{node}']\}}$ (**cons**)
$\{(\forall \mathbf{node}, \mathbf{fid}(\mathbf{tag}_{\mathbf{node}}[\emptyset]_{\mathtt{fid}}) \triangleright P[\mathbf{node}/\mathbf{node}'])(\emptyset)\}$
$\mathbf{node}' = \text{createNode}(\mathbf{tag});$
$\{P\}$

5.3.2 getNodeName

$\{\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}}\}$
$\mathbf{tag}' = \text{getNodeName}(\mathbf{node});$
$\underline{\{\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}} \wedge \mathbf{tag}' = \mathbf{tag}\}}$ (**frame**)
$\{(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}} \triangleright P[\mathbf{tag}/\mathbf{tag}'])(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}})\}$
$\mathbf{tag}' = \text{getNodeName}(\mathbf{node});$
$\underline{\{(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}} \triangleright P[\mathbf{tag}/\mathbf{tag}'])(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}} \wedge \mathbf{tag}' = \mathbf{tag})\}}$ (**cons**)
$\{(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}} \triangleright P[\mathbf{tag}/\mathbf{tag}'])(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}})\}$
$\mathbf{tag}' = \text{getNodeName}(\mathbf{node});$
$\underline{\{(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}} \triangleright P[\mathbf{tag}/\mathbf{tag}'])(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}}) \wedge (\mathbf{tag}' = \mathbf{tag})\}}$ (**cons**)
$\{\mathbf{True}(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}}) \wedge P[\mathbf{tag}/\mathbf{tag}']\}$
$\mathbf{tag}' = \text{getNodeName}(\mathbf{node});$
$\underline{\{P[\mathbf{tag}/\mathbf{tag}'] \wedge (\mathbf{tag}' = \mathbf{tag})\}}$ (**cons**/**aux**)
$\{\exists \mathbf{tag}(\mathbf{True}(\mathbf{tag}_{\mathbf{node}}[\mathbf{F}]_{\mathtt{fid}}) \wedge P[\mathbf{tag}/\mathbf{tag}'])\}$
$\mathbf{tag}' = \text{getNodeName}(\mathbf{node});$
{P}

5.3.3 getChildNodes

$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{parent});$
$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{forest} = \mathbf{fid}\}$ (**frame**)

$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathbf{fid}/\mathbf{forest}])(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}})\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{parent});$
$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathbf{fid}/\mathbf{forest}](\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{forest} = \mathbf{fid})\}$ (**cons**)

$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathbf{fid}/\mathbf{forest}])(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}})\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{parent});$
$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathbf{fid}/\mathbf{forest}](\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}) \wedge (\mathbf{forest} = \mathbf{fid})\}$ (**cons**)

$\{\mathbf{True}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}) \wedge P[\mathbf{fid}/\mathbf{forest}]\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{parent});$
$\{P[\mathbf{fid}/\mathbf{forest}] \wedge (\mathbf{forest} = \mathbf{fid})\}$ (**cons**/**aux**)

$\{\exists \mathbf{fid}(\mathbf{True}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathbf{fid}}) \wedge P[\mathbf{fid}/\mathbf{forest}])\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{parent});$
$\{P\}$

5.3.4 getParentNode

$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}}\}$
$\mathbf{parent}' = \text{getParentNode}(\mathbf{child});$
$\{\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathbf{parent}' = \mathbf{parent}\}$ (**frame**)

$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}} \triangleright P[\mathbf{parent}/\mathbf{parent}'])$
$(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}})\}$
$\mathbf{parent}' = \text{getParentNode}(\mathbf{child});$
$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}} \triangleright P[\mathbf{parent}/\mathbf{parent}'])$
$(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathbf{parent}' = \mathbf{parent})\}$ (**cons**)

$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}} \triangleright P[\mathbf{parent}/\mathbf{parent}'])$
$(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}})\}$
$\mathbf{parent}' = \text{getParentNode}(\mathbf{child});$
$\{(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}} \triangleright P[\mathbf{parent}/\mathbf{parent}'])$
$(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}}) \wedge (\mathbf{parent}' = \mathbf{parent})\}$ (**cons**)

$\{\mathbf{True}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}}) \wedge P[\mathbf{parent}/\mathbf{parent}']\}$
$\mathbf{parent}' = \text{getParentNode}(\mathbf{child});$
$\{P[\mathbf{parent}/\mathbf{parent}'] \wedge (\mathbf{parent}' = \mathbf{parent})\}$ (**cons**/**aux**)

$\{\exists \mathbf{parent}(\mathbf{True}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F}_2]_{\mathbf{fid}}) \wedge P[\mathbf{parent}/\mathbf{parent}'])\}$
$\mathbf{parent}' = \text{getParentNode}(\mathbf{child});$
$\{P\}$

5.3.5 insertBefore

$$
\begin{array}{l}
\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))(\mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''})\} \\
\text{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}); \\
\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}})\} \qquad (\mathbf{frame}) \\
\hline
\{(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}) \triangleright P) \\
\quad ((\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))(\mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''}))\} \\
\text{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}); \\
\{(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}) \triangleright P) \\
\quad (\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))\} \qquad (\mathbf{cons}) \\
\hline
\{(\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}) \triangleright P) \\
\quad ((\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))(\mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}''}))\} \\
\text{insertBefore}(\mathbf{parent}, \mathbf{newChild}, \mathbf{ref}); \\
\{P\}
\end{array}
$$

5.3.6 removeChild

$$
\begin{array}{l}
\{\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}})\} \\
\mathbf{node} = \text{removeChild}(\mathbf{parent}, \mathbf{child}); \\
\{\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \wedge \mathbf{node} = \mathbf{child}\} \qquad (\mathbf{frame}) \\
\hline
\{(\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \triangleright P[\mathbf{child}/\mathbf{node}]) \\
\quad (\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))\} \\
\mathbf{node} = \text{removeChild}(\mathbf{parent}, \mathbf{child}); \\
\{(\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \triangleright P[\mathbf{child}/\mathbf{node}]) \\
\quad (\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \wedge \mathbf{node} = \mathbf{child})\} \qquad (\mathbf{cons}) \\
\hline
\{(\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \triangleright P[\mathbf{child}/\mathbf{node}]) \\
\quad (\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))\} \\
\mathbf{node} = \text{removeChild}(\mathbf{parent}, \mathbf{child}); \\
\{(\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \triangleright P[\mathbf{child}/\mathbf{node}]) \\
\quad (\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'}) \wedge (\mathbf{node} = \mathbf{child})\} \qquad (\mathbf{cons}) \\
\hline
\{(\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \triangleright P[\mathbf{child}/\mathbf{node}]) \\
\quad (\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))\} \\
\mathbf{node} = \text{removeChild}(\mathbf{parent}, \mathbf{child}); \\
\{P[\mathbf{child}/\mathbf{node}] \wedge (\mathbf{node} = \mathbf{child})\} \qquad (\mathbf{cons/aux}) \\
\hline
\{\exists \mathbf{child}(\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{F}_2]_{\mathfrak{fid}}) \oplus \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \triangleright P[\mathbf{child}/\mathbf{node}]) \\
\quad (\mathbf{CT}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag}'_{\mathbf{child}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F}_2]_{\mathfrak{fid}}))\} \\
\mathbf{node} = \mathit{removeChild}(\mathbf{parent}, \mathbf{child}); \\
\{P\}
\end{array}
$$

5.3.7 getLength

$$
\begin{array}{l}
\{\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}}\} \\
\mathbf{length} = \mathrm{getLength}(\mathbf{fid}); \\
\hline
\{\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{length} = \mathrm{lng}(\mathbf{F})\} \qquad (\mathbf{frame}) \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathrm{lng}(\mathbf{F})/\mathbf{length}])(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}})\} \\
\mathbf{length} = \mathrm{getLength}(\mathbf{fid}); \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathrm{lng}(\mathbf{F})/\mathbf{length}])(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{length} = \mathrm{lng}(\mathbf{F}))\} \qquad (\mathbf{cons}) \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathrm{lng}(\mathbf{F})/\mathbf{length}])(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}})\} \\
\mathbf{length} = \mathrm{getLength}(\mathbf{fid}); \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}} \triangleright P[\mathrm{lng}(\mathbf{F})/\mathbf{length}])(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}}) \wedge (\mathbf{length} = \mathrm{lng}(\mathbf{F}))\} \qquad (\mathbf{cons}) \\
\hline
\{\mathbf{True}(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}}) \wedge P[\mathrm{lng}(\mathbf{F})/\mathbf{length}]\} \\
\mathbf{length} = \mathrm{getLength}(\mathbf{fid}); \\
\hline
\{P[\mathrm{lng}(\mathbf{F})/\mathbf{length}] \wedge (\mathbf{length} = \mathrm{lng}(\mathbf{F}))\} \qquad (\mathbf{cons/aux}) \\
\hline
\{\exists \mathbf{F}(\mathbf{True}(\mathbf{tag_{node}}[\mathbf{F}]_{\mathbf{fid}}) \wedge P[\mathrm{lng}(\mathbf{F})/\mathbf{length}])\} \\
\mathbf{length} = \mathrm{getLength}(\mathbf{fid}); \\
\{P\}
\end{array}
$$

5.3.8 getItem

$$
\begin{array}{l}
\{\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length}\} \\
\mathbf{node} = \mathrm{getItem}(\mathbf{length}, \mathbf{fid}); \\
\hline
\{\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \wedge \mathbf{node} = \mathbf{item}\} \qquad (\mathbf{frame}) \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \triangleright P[\mathbf{item}/\mathbf{node}]) \\
\quad (\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length})\} \\
\mathbf{node} = \mathrm{getItem}(\mathbf{length}, \mathbf{fid}); \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \triangleright P[\mathbf{item}/\mathbf{node}]) \\
\quad (\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \wedge \mathbf{node} = \mathbf{item})\} \qquad (\mathbf{cons}) \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \triangleright P[\mathbf{item}/\mathbf{node}]) \\
\quad (\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length})\} \\
\mathbf{node} = \mathrm{getItem}(\mathbf{length}, \mathbf{fid}); \\
\hline
\{(\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length} \triangleright P[\mathbf{item}/\mathbf{node}]) \\
\quad (\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length}) \wedge (\mathbf{node} = \mathbf{item})\} \qquad (\mathbf{cons}) \\
\hline
\{\mathbf{True}(\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length}) \wedge P[\mathbf{item}/\mathbf{node}]\} \\
\mathbf{node} = \mathrm{getItem}(\mathbf{length}, \mathbf{fid}); \\
\hline
\{P[\mathbf{item}/\mathbf{node}] \wedge (\mathbf{node} = \mathbf{item})\} \qquad (\mathbf{cons/aux}) \\
\hline
\{\exists \mathbf{item}(\mathbf{True}(\mathbf{tag_{node}}[\mathbf{F}_1 \times \mathbf{tag'_{item}}[\mathbf{F'}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}} \wedge \mathrm{lng}(\mathbf{F}_1) = \mathbf{length}) \wedge P[\mathbf{item}/\mathbf{node}])\} \\
\mathbf{node} = \mathrm{getItem}(\mathbf{length}, \mathbf{fid}); \\
\{P\}
\end{array}
$$

It is easy to verify that the pre-conditions derived above are indeed the weakest pre-conditions for these commands.

6 Building Simple Programs

So far we have specified the entire behavior of Minimal DOM. In this section we will show that we can use the specification of Minimal DOM to generate the specification for the structural behaviour of DOM Core Level 1. Namely we will show that we can now specify the behaviour of the commands "replaceChild", "cloneNode" and "hasChildNodes". It is the compositional power of the Sequencing Rule that allows us to build up the specification axioms for these new commands. Since we have already derived the weakest per-conditions for the commands of Minimal DOM, we know that these commands are safe. This safety property can be carried over to the new commands so long as they are specified using just the commands of Minimal DOM. As we shall show in this section, we are able to do just this.

6.1 replaceChild

The replaceChild method operates on some **parent** Node and takes two other Nodes as arguments, **newChild** and **oldChild**. It replaces the Node **oldChild** with the **newChild** Node in the list of children of **parent** and returns a pointer to **oldChild** which is moved to the top level of the DOM grove. If **newChild** is already in the tree, it is first removed. As with all of our methods, we pass the Node the method is called upon explicitly as the first argument. We can build this command from the commands of Minimal DOM as illustrated in figure 12 and as described below:

```
node = replaceChild(parent, newChild, oldChild){

                    insertBefore(parent, newChild, oldChild);
                    node = removeChild(parent, oldChild);

}
```

We specify the Axiom for this command as:

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag'}_{\mathbf{oldChild}}[\mathbf{F}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}}))(\mathbf{tag''}_{\mathbf{newChild}}[\mathbf{F'}]_{\mathbf{fid''}})\}$
$\mathbf{node}$ = replaceChild(**parent**, **newChild**, **oldChild**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag''}_{\mathbf{newChild}}[\mathbf{F'}]_{\mathbf{fid''}} \times \mathbf{F}_2]_{\mathbf{fid}}) \oplus \mathbf{tag'}_{\mathbf{oldChild}}[\mathbf{F}]_{\mathbf{fid'}} \wedge \mathbf{node} = \mathbf{oldChild}\}$

We prove this by taking the pre-condition above and then, by applying the Local Hoare Triple axioms of our existing commands, show that we can get to the post-condition given above. The Sequencing Rule of our reasoning framework gives us the power to make these logical steps, although some use of the Consequence Rule may be necessary to take the post-condition of one command to the correct form to be the pre-condition of the next command.

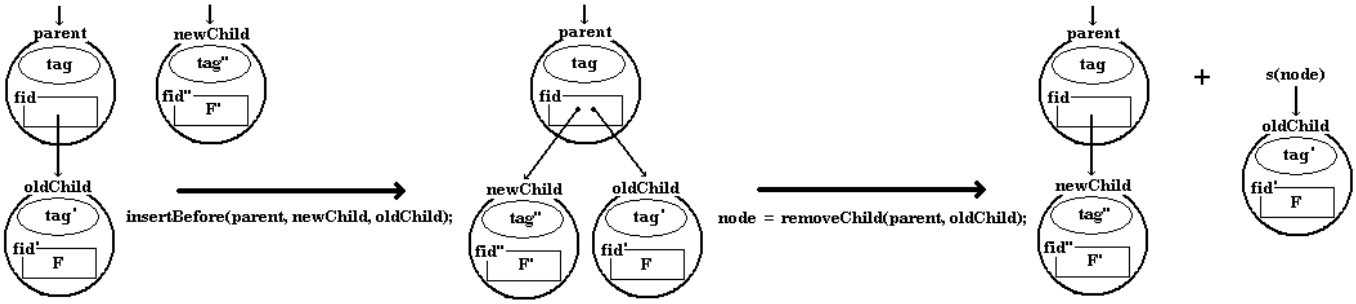

Figure 12: replaceChild(parent, newChild, oldChild)

So the proof for replaceChild goes as follows:

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag'}_{\mathbf{oldChild}}[\mathbf{F}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}}))(\mathbf{tag''}_{\mathbf{newChild}}[\mathbf{F'}]_{\mathbf{fid''}})\}$
insertBefore(**parent**, **newChild**, **oldChild**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag''}_{\mathbf{newChild}}[\mathbf{F'}]_{\mathbf{fid''}} \times \mathbf{tag'}_{\mathbf{oldChild}}[\mathbf{F}]_{\mathbf{fid'}} \times \mathbf{F}_2]_{\mathbf{fid}})\}$
node = removeChild(**parent**, **oldChild**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}_1 \times \mathbf{tag''}_{\mathbf{newChild}}[\mathbf{F'}]_{\mathbf{fid''}} \times \mathbf{F}_2]_{\mathbf{fid}}) \oplus \mathbf{tag'}_{\mathbf{oldChild}}[\mathbf{F}]_{\mathbf{fid'}} \ \wedge \ \mathbf{node} = \mathbf{oldChild}\}$

6.2 cloneNode

The cloneNode command returns a duplicate of the Node it is called upon. For our purposes this means that the new Node will have the same **tag** as the old Node. The command is not a deep clone, so it does not copy the Node's children or any of the subtree beneath it. The new Node will have different, fresh, identifiers for both its Node and list identifiers. The duplicate Node has no parent, i.e. it is created at the top level in our grove. We can build this command from the commands of Minimal DOM as shown in figure 13 which is as follows:

```
node = cloneNode(id){

                    tag' = getNodeName(id);
                    node = createNode(tag');

}
```

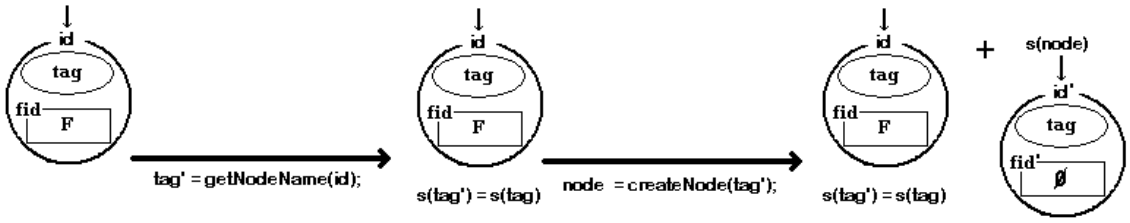

Figure 13: cloneNode(id)

We specify the Axiom for this command as:

$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{node}$ = cloneNode($\mathbf{id}$);
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \oplus \mathbf{tag_{id'}}[\emptyset]_{\mathbf{fid'}} \wedge \mathbf{node} = \mathbf{id'}\}$

Using the same style of proof as set out for the replaceChild command, we can prove that this axiom is correct thusly:

$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{tag'}$ = getNodeName($\mathbf{id}$);
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{tag'} = \mathbf{tag}\}$
$\mathbf{@n}$ = createNode($\mathbf{tag'}$);
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \oplus \mathbf{tag'_{id'}}[\emptyset]_{\mathbf{fid'}} \wedge \mathbf{tag'} = \mathbf{tag} \wedge \mathbf{node} = \mathbf{id'}\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \oplus \mathbf{tag_{id'}}[\emptyset]_{\mathbf{fid'}} \wedge \mathbf{node} = \mathbf{id'}\}$

6.3 hasChildNodes

The hasChildNodes command is the last command we need to consider to fully specify the structural behaviour of DOM Core Level 1. It is a convenience method which allows quick determination of whether a Node has any children. Note that this program uses boolean tests which we included in our logic definition, but have not used until this point. We can build this command from the commands of Minimal DOM as follows:

bool = hasChildNodes(id){

forest = getChildNodes(id);
length = getLength(forest);
bool = not(length = 0);

}

Since there is no manipulation of the DOM grove in this command, we do not feel that a diagram adds to the explanation of the command. The Axiom for this command is specified as:

$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{bool} = \text{hasChildNodes}(\mathbf{id});$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge ((\text{lng}(\mathbf{F}) = 0 \wedge \mathbf{bool} = \text{false}) \vee (\text{lng}(\mathbf{F}) \neq 0 \wedge \mathbf{bool} = \text{true}))\}$

Note that the length of a list cannot be a negative number, so the $\neq 0$ condition is sufficient to express > 0. There are two cases for the proof, which we give below:

Case 1: $\mathbf{F} = \emptyset$

$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{fid}}\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{id});$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{fid}} \wedge \mathbf{forest} = \mathbf{fid}\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid}\}$
$\mathbf{length} = \text{getLength}(\mathbf{forest});$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} = \text{lng}(\emptyset)\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} = 0\}$
$\mathbf{bool} = \text{not}(\mathbf{length} = 0);$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} = 0 \wedge \mathbf{bool} = \neg(\mathbf{length} = 0)\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{bool} = \neg(0 = 0)\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{bool} = \neg(\text{true})\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{bool} = \text{false}\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{fid}} \wedge \mathbf{bool} = \text{false}\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{fid}} \wedge (\text{lng}(\emptyset) = 0 \wedge \mathbf{bool} = \text{false})\}$
$\{\mathbf{tag_{id}}[\emptyset]_{\mathbf{fid}} \wedge ((\text{lng}(\emptyset) = 0 \wedge \mathbf{bool} = \text{false}) \vee (\text{lng}(\emptyset) \neq 0 \wedge \mathbf{bool} = \text{true}))\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge ((\text{lng}(\mathbf{F}) = 0 \wedge \mathbf{bool} = \text{false}) \vee (\text{lng}(\mathbf{F}) \neq 0 \wedge \mathbf{bool} = \text{true}))\}$

Case 2: $\mathbf{F} \neq \emptyset$

$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}}\}$
$\mathbf{forest} = \text{getChildNodes}(\mathbf{id});$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{forest} = \mathbf{fid}\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid}\}$
$\mathbf{length} = \text{getLength}(\mathbf{forest});$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} = \text{lng}(\mathbf{F})\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} \neq 0\}$
$\mathbf{bool} = \text{not}(\mathbf{length} = 0);$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} \neq 0 \wedge \mathbf{bool} = \neg(\mathbf{length} = 0)\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{forest}} \wedge \mathbf{forest} = \mathbf{fid} \wedge \mathbf{length} \neq 0 \wedge \mathbf{bool} = \text{true}\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge \mathbf{length} \neq 0 \wedge \mathbf{bool} = \text{true}\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge (\text{lng}(\mathbf{F}) \neq 0 \wedge \mathbf{bool} = \text{true})\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge ((\text{lng}(\mathbf{F}) = 0 \wedge \mathbf{bool} = \text{false}) \vee (\text{lng}(\mathbf{F}) \neq 0 \wedge \mathbf{bool} = \text{true}))\}$
$\{\mathbf{tag_{id}}[\mathbf{F}]_{\mathbf{fid}} \wedge ((\text{lng}(\mathbf{F}) = 0 \wedge \mathbf{bool} = \text{false}) \vee (\text{lng}(\mathbf{F}) \neq 0 \wedge \mathbf{bool} = \text{true}))\}$

6.4 DOM Core Level 1 - Evaluation

We have shown that we can give a complete specification for the structural behaviour of DOM Core Level 1 from the specification of Minimal DOM. This should be evidence enough to justify our earlier claim that Minimal DOM is indeed sufficient to provide the structural specification that we desire. The derivation of the weakest pre-conditions shows that this specification is safe, so we do indeed now have a more formal specification that could be the basis of a verification engine, at least over commands that are only concerned with the structure of a DOM grove. The actual implementation of such a verification engine falls outside of the scope of this project, but with our strong reasoning framework in place it should now be a relatively easy task.

7 Going Further

Showing that we can derive the structural behaviour of DOM Core Level 1 from Minimal DOM, whilst useful, does not demonstrate the true power of our specification. We shall show that even some commands not specified in DOM Core Level 1 can still be easily specified from the commands of Minimal DOM using the same techniques as set out above in section 6.

7.1 insertAfter

As a simple example let us consider a command which behaves much like insertBefore, only the **newChild** Node is inserted into the list of **parent**'s children after the Node **ref** rather than before it. Figure 14 shows how we can construct this command using Minimal DOM, namely that we have the following,

insertAfter(parent, newChild, ref){

insertBefore(parent, newChild, ref);
insertBefore(parent, ref, newChild);

}

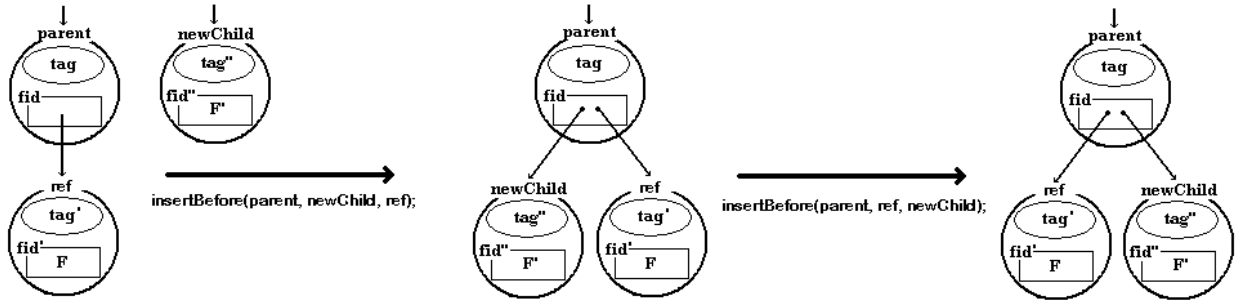

Figure 14: insertAfter(parent, newChild, ref)

Similar to the axiom for insertBefore, the axiom that we would expect for insertAfter is:

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F_1} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{F_2}]_{\mathbf{fid}}))(\mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}''})\}$
insertAfter(**parent**, **newChild**, **ref**);
$\{\mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F_1} \times \mathbf{tag}'_{\mathbf{ref}}[\mathbf{F}]_{\mathbf{fid}'} \times \mathbf{tag}''_{\mathbf{newChild}}[\mathbf{F}']_{\mathbf{fid}''} \times \mathbf{F_2}]_{\mathbf{fid}})\}$

Using the Local Hoare Triple axiom for insertBefore in a sequential manor we can show that the above Triple is indeed correct for the insertAfter command. The proof is as follows:

$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F_2}]_{\mathfrak{fid}}))(\mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''})\}$
insertBefore(**parent**, **newChild** , **ref**);
$\{\mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F_2}]_{\mathfrak{fid}})\}$
$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \emptyset \times \mathbf{F_2}]_{\mathfrak{fid}}))(\mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'})\}$
$\{(\emptyset \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{F_2}]_{\mathfrak{fid}}))(\mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'})\}$
insertBefore(**parent**, **ref**, **newChild**);
$\{\mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{F_2}]_{\mathfrak{fid}})\}$

The applications of the insertBefore axiom are trivial, but producing the correct pre-condition for the second call to insertBefore is a bit trickier. Let us look at these steps in more detail. After we have called insertBefore for the first time we have the post-condition,

$$\mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'} \times \mathbf{F_2}]_{\mathfrak{fid}})$$

Now this formula give us more information than we require for the pre-condition of the next insertBefore command. Not only do we have that the Nodes **parent**, **newChild** and **ref** exist where **ref** is not above **parent**, but we also know exactly where **ref** is. In order to get this into the correct from we need to loose information. So we split apart the subtree beginning at **ref** from the rest of the tree with the context,

$$(\emptyset \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \emptyset \times \mathbf{F_2}]_{\mathfrak{fid}}))$$

Here we state that if we replace **ref** with $\emptyset$ then the grove satisfies,

$$\mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \emptyset \times \mathbf{F_2}]_{\mathfrak{fid}})$$

but this is equivalent to,

$$\mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{F_2}]_{\mathfrak{fid}})$$

Hence we are able to conclude that our grove satisfies,

$$(\emptyset \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F_1} \times \mathbf{tag''_{newChild}}[\mathbf{F}']_{\mathfrak{fid}''} \times \mathbf{F_2}]_{\mathfrak{fid}}))(\mathbf{tag'_{ref}}[\mathbf{F}]_{\mathfrak{fid}'})$$

which is in the correct form for us to be able to apply the axiom for the insertBefore command once again.

7.2 Equality

If we want to provide some kind of tree equality test, that checks if two trees have the same structure and data content, we would need to introduce some form of recursion into our reasoning. However, the case of testing if two nodes contain the same data, i.e. have the same **tag**, is simpler and can be specified directly from our current specification for Minimal DOM. We would need

a command of the form,

bool = nodeEquality(id1, id2){

tag = getNodeName(id1);
tag' = getNodeName(id2);
bool = (tag = tag')

}

We would expect the Local Hoare Triple for this command to be as follows:

$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})\}$
$\mathbf{bool} = \text{nodeEquality}(\mathbf{id1}, \mathbf{id2});$
$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})$
$\wedge((\mathbf{data1} = \mathbf{data2}) \wedge (\mathbf{bool} = \text{true})) \vee ((\mathbf{data} \neq \mathbf{data2}) \wedge (\mathbf{bool} = \text{false}))\}$

We can prove that this Local Hoare Triple is correct in the following fashion:

$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})\}$
$\mathbf{tag} = \text{getNodeName}(\mathbf{id1});$
$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'}) \wedge (\mathbf{tag} = \mathbf{data1})\}$
$\mathbf{tag}' = \text{getNodeName}(\mathbf{id2});$
$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'}) \wedge (\mathbf{tag} = \mathbf{data1}) \wedge (\mathbf{tag}' = \mathbf{data2})\}$
$\mathbf{bool} = (\mathbf{tag} = \mathbf{tag}');$
$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})$
$\wedge (\mathbf{tag} = \mathbf{data1}) \wedge (\mathbf{tag}' = \mathbf{data2}) \wedge (\mathbf{bool} = (\mathbf{tag} = \mathbf{tag}'))\}$
$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})$
$\wedge (\mathbf{bool} = (\mathbf{data1} = \mathbf{data2}))\}$
$\{\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})$
$\wedge((\mathbf{bool} = \text{true}) \wedge (\mathbf{data1} = \mathbf{data2})) \vee (((\mathbf{bool} = \text{false}) \wedge (\mathbf{data1} \neq \mathbf{data2})))\}$

Note that in the final step, rather than giving two separate proofs, we have combined the information in the obvious fashion to arrive at our desired post-condition. We could have laid out this proof over two cases, one where **data1** = **data2** and one where **data1** $\neq$ **data2**, but we felt that this only served to delay an already obvious conclusion.

One final point to make is the meaning of the condition,

$$\mathbf{CG}(\mathbf{data1}_{\mathbf{id1}}[\mathbf{F}]_{\mathbf{fid}}) \wedge \mathbf{CG}'(\mathbf{data2}_{\mathbf{id2}}[\mathbf{F}']_{\mathbf{fid}'})$$

The $\wedge$ simply specifies that both of the existence properties of the grove are true. Namely that both the nodes **id1** and **id2** exist somewhere in the grove. Note that in this case we do not require that **id2** is not an ancestor of **id1**.

8 Evaluation

We have generally provided details of our work in a progressive story-like fashion, making adjustments as we are going and providing justification for our choices as well as some brief evaluation of our results. However, in the following section we will pull all of this together to give an overall evaluation of the success of the project. We also discuss the various possibilities for future work that follow on naturally from this project.

8.1 Success of the Project

When we talk about the success of the project we need to be careful to remember that we had several different aims in mind throughout this project. As such it seems only sensible to address these in turn. Before we try to sum up the specific successes and failings of our project, we will first consider the two broader, more general aims. Specifically why our work matters to the DOM and W3C communities. After this we shall move on to a more detailed analysis of what our logical specification has provided.

8.1.1 Pleasing the DOM Community

As our work has progressed we have always been careful to make the same choices that DOM currently makes, or stay as close to DOM as we can. However this is a tricky task as the very nature of the current DOM specification is to only specify what they have to. Such details as the data structure are left to the user to define. If we had stuck with such a style all the way, we would never have been able to get anywhere with our reasoning. We are confident that our specification is correct, but it may not match up to exactly what the DOM community may expect. It is at this point we have to make a hard choice. Do we stick to our guns and try to get the DOM community to adapt their way of thinking to our specification, or do we have to accept that this might not happen and try to change our specification. Our suspicion is that we need to do a little of both. Clearly we can't expect to change the way DOM works in one fell swoop, but we can hope to show the DOM community how perhaps they should be thinking of DOM. Our specification leads itself to easy extension, and automated verification, two big things that the current specification simply cannot do. However, we have sacrificed some of the ease of understanding that the English specification provides. However, we feel that the removal of ambiguities from the specification is well worth this cost. So do we think the DOM community will be pleased? Well, as with all communities the answer will be a mixture of yes and no. Some will herald the introduction of formalism to the world of DOM, but equally others might argue that everything we've have done is already there. Perhaps true, but we are confident that the clarity and possibilities that a formal specification bring forth will win through in time.

8.1.2 Pleasing the W3C Community

The W3C (World Wide Web Consortium) Community, on the other hand, ought to be pleased as a whole with the specification we have produced. The W3C organisations principle aim is to 'develop interoperable technologies (specifications, guidelines, software, and tools) to lead the Web to its full potential'. We have shown that our specification for Minimal DOM is highly expressive, highly formal, and highly extendable. These are exactly the kinds of properties that the W3C was set up to try and bring into commonplace web applications. So long as we can maintain these properties throughout our future work to bring our specification up to the whole of DOM, then we are confident that the W3C community will be very interested in our final product. The only criticism we might expect from the W3C community is that our work is still very much a shadow of an object orientated implementation of DOM. Although we have done our best to keep our reasoning platform free, our specification still makes some key object orientated assumptions. In future it would make more sense to specify XML update at the XML level.

8.1.3 The Data Structure

The Grove, Tree and Forest data structure that we have chosen to operate over could, by some, be deemed over complicated or contrived. However, we believe that this is far from the case. In our search for a suitable data structure we were unable to find a simpler structure that did not add significant difficulties to our logic. It is the very nature of the top level, or grove level as we choose to call it, of DOM which causes the problems with other, simpler, data structures. The fact that this top set is unordered, where as the list of children of a node is ordered, leads to a necessary difference in composition operator. To define this over single inductive data type means that we would need to carry around a notion of tree depth in order to know when our tree compositions are commutative and when they are not. Using the three typed structure set out by Smith [8] implicitly encodes this information into the data structure in a simple and neat fashion.

Another point that could come under some criticism is the choice of overloading the notation of $\emptyset$ to be both the empty set and the empty list. Again though we are able to defend our choice based on the complexities that arise from any other approach. Clearly, if we choose to treat the empty set and empty list as separate entities then we will have to deal with them separately as well. This would cause a great deal of problems when we get to defining our Local Hoare Triples. Instead of the one neat, if slightly complicated, pre-condition we currently have for our move commands,

$$(\emptyset \triangleright \mathbf{CG}(\mathbf{tag}_{\mathbf{parent}}[\mathbf{F}]_{\mathfrak{fid}}))(\mathbf{tag}'_{\mathbf{newChild}}[\mathbf{F}']_{\mathfrak{fid}'})$$

we would instead have the disjunction of two equally complicated formulae.

$$(\emptyset_g \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F}]_{\mathbf{fid}}))(\mathbf{tag'_{newChild}}[\mathbf{F'}]_{\mathbf{fid'}}) \vee (\emptyset_f \triangleright \mathbf{CG}(\mathbf{tag_{parent}}[\mathbf{F}]_{\mathbf{fid}}))(\mathbf{tag'_{newChild}}[\mathbf{F'}]_{\mathbf{fid'}})$$

The first half here specifying that the hole can be at the top level, the other specifying that it can be within a forest somewhere. Not only does this lead to much more complex notation, but it also makes the composition of our Axioms much harder than it is at current. So from a practical point of view our choice makes sense and so it does logically too. Both the empty set and the empty list describe what it means to have nothing. But nothing is the same, no matter what kind of nothing it is. Only when we add something to nothing can we get an effect, but at this point we can pick the nothing that fits with the added data type. This, in essence, is what our overloading of notation achieves.

A final point to address, that was raised by a discussion with Ian Hodkinson, is why we have chosen to work with a list composition operator $\times$ instead of the more widespread : operator as seen in many languages such as Haskell. The issue with the latter approach, where we view lists as **head**:**tail**, is how we represent items at an arbitrary position of the list. No notion of **list**:**element**:**list** exists in this new data structure, we would have to explicitly list out all of the elements of the first list. This makes the use of context's untenable, much as in Cheung's work [6] due to the inability to express the arbitrary positioning of context holes without significantly more reasoning effort. In effect, Haskell's view of a list is closer to the node level sibling relation than it is to our, higher level, reasoning. The same reasons that caused us to move away from Cheung's style of reasoning are behind our decision to work with our list structure instead of another form of list representation.

In summary, we believe that the data structure we have provided is both the simplest that is possible, and also utilises a clear form of notation that lends itself to more readable Axioms for our commands.

8.1.4 The Logical Framework

The logical framework that we provide for our specification is one of the most important features of this project. It is the framework that allows us to build up bigger commands from our commands of Minimal DOM and derive their specification without having to directly compute their weakest pre-conditions. It is because these bigger commands are built from our Minimal DOM commands that we can instantly deduce that the new commands will also be safe. If a new command were not safe, then some step of the command must not be safe, but all of the steps of the new command are provably safe commands from Minimal DOM. Therefore the new command must also be safe. The compositional construction of new commands that our logical framework allows lets us derive more than just DOM Core Level 1, as we saw above, so our specification is far more expressive than the existing one. It also lends itself naturally to

extensions, which makes it far more useful than the current specification for the purposes of understanding how one might extend DOM itself, a task that many programmers find tricky with the current specification.

8.1.5 The Axioms

The one failing of our work to date is that the axioms for our commands are not all 'Small'. Specifically, the move commands, append, insertBefore and removeChild, all have footprints that can be arbitrarily large as we discussed before. Some work has gone into producing a pre-condition which ensures that the footprint is as small as possible. A formula of the form,

$$\wedge\neg((\neg\_)(\emptyset \triangleright \mathrm{true}(\mathrm{tag}_{\mathrm{parent}}[F]_{\mathrm{fid}}))(\mathrm{tag'}_{\mathrm{newChild}}[F']_{\mathrm{fid'}}))$$

can be added to the current pre-condition. The example above is for the append command. The formula inside the outer $\neg$ describes a grove where the context around the **parent** Node must include **newChild** and also be bigger than this. Adding the extra $\neg$ around this formula stops the grove being any bigger than it needs to be to include both the **parent** and **newChild** Nodes. However, there are two main issues with this approach. The first is that the readability of our axioms is significantly reduced with this extra condition added to our axioms. Whilst this is not a logical problem, we must remember that we are trying to make our specification readable by people who may not have a huge amount of experience with logic. In order to achieve this we need to try and keep our axioms as notationally simple as possible. The second issue is that it is tricky to derive this condition when composing axioms of Minimal DOM together. Since we have already seen how powerful composing our commands can be, we do not want to throw this away. So, at present, we do not have a neat solution to this problem.

8.1.6 Conclusion

Overall we have provided more than we set out to achieve in the scope of this project. We had hoped to be able to specify the structure of Minimal DOM, but in fact we have shown that we are able to specify much more than that from our specification for Minimal DOM. The framework we have defined should also be easily extendable to higher levels of DOM without loosing any of the clarity it currently possesses. All in all we believe that our new specification is far more precise and generally useful than the current specification and shows great promise for being scaled up to cover the whole of DOM.

8.2 Future Work

There are many possible directions that we could take this work in future as we have created a very expressive and powerful logical framework for the structural

behaviour of DOM. However there are also a few issues that we have not completely resolved in the course of this project that should be correctly addressed. We shall briefly discuss the possible future extensions that we believe to be of greatest interest, or greatest necessity, to the DOM, W3C and Context Logic communities.

8.2.1 Verification Tool

The most obvious extension to this project would be the provision of an automated verification tool for DOM Core Level 1. This would be a program that is capable of taking some DOM program, which uses just the commands from DOM Core Level 1, and being able to check that the program behaves as expected. Providing the pre- and post-conditions of the program, we can see if these match up to our expected program behaviour. We have already provided a complete specification for all of the structural behaviour, and there is only a small amount of detail on top of this in the Core Level. As such, we would hope that the task of creating such a verification tool should now be fairly easy.

8.2.2 Specifying More of DOM

Whilst having specified all of DOM Core Level 1 is an important achievement in our goal for a fully formally specified DOM, it is far from the whole story. There are two more levels of DOM, DOM level 2 and DOM level 3, as well as DOM HTML Level 1, all of which will need to be specified. Some of the work will be trivial extensions of Minimal DOM. As we have shown, Minimal DOM can express more than just DOM Core level 1, and its compositional nature makes extending the specification to larger programs a fairly straight forward task. However, at some point it is likely that we will find concepts that we have not had to consider at the Core Level, and this will require new additions to the specification. Identifying these concepts, specifying them and combining this work with Minimal DOM will bring us even closer, perhaps all the way, to a complete formal specification for the whole of DOM.

8.2.3 Recursion

The current Minimal DOM specification is limited in the sense that it has no definition for any kind of looping program structure. As we noted above, this puts a limit on the kinds of programs that we are currently able to specify. Whilst it could be argued that an investigation of recursive techniques falls under the previous category of 'Specifying More of DOM', we feel that this case deserves special consideration. Adding the power of recursion to our our specification will open up a whole new branch of interesting programs that we will be able to specify. Also picking different styles of recursion, while loops, do-until loops or for loops for example, will produce different styles of DOM and limit which programming languages interfaces our specification will be able to

handle. The study of this will provide valuable insights into how best to tackle specifying DOM for all of its different implementations, as we will need to be able to specify different language implementations which are not guaranteed to have the same program structures available. We also believe there may be some interesting logical results about how different recursive styles can provide identical or vastly different outputs.

8.2.4 Small Axioms

As we have discussed before, our move commands do not currently have Axioms that are 'Small'. Whilst we have put the issue of the size of the command's footprints to one side for the scope of this project, it is a key point that needs to be addressed before the specification can hope to become mainstream. It is possible that the 'Medium Axioms' we have given will turn out to be sufficient, but we find this unlikely. A small amount of work has already gone into looking for a different way to express the 'not an ancestor' property that our move commands require, but nothing satisfactory gas come out of this work as of yet. We believe that a concerted effort should be made to investigate these axioms and come to a conclusion one way or the other.

8.2.5 Multi-holed Contexts

There is currently a great deal of work going in to investigating the definition of multi-holed contexts, that is contexts that are able to be applied to more than one piece of data simultaneously. The possible utility of these contexts to our data structure should be investigated as they may provide a possible solution to our large footprint issue for the move commands of Minimal DOM. It is also possible that they may lead to neater Local Hoare Triples in general. However, until this work is completed, we will have to put this particular avenue of research on hold.

8.2.6 Concurrent DOM

So far we have only considered the specification of serial XML update, but there are several concurrent DOM implementations already in existence. These implementations all conform to the serial DOM specification, but there is no concurrent DOM specification. The English language is not able to accurately express the complex word of concurrency, but a more formal specification, such as ours, should be up to the challenge. O'Hearn has already produced work on concurrent reasoning for Separation Logic. Bringing similar results to the world of Context Logic would be a crucial step in providing a Context Logic reasoning about concurrent tree update.

9 References

1. Bornat, R., Calcagno, C., O'Hearn, P.: Local reasoning, separation and aliasing. SPACE(2004)
2. Reynold, J.: Separation Logic: A Logic for Shared Mutable Data Structures. IEEE(2002)
3. Calcagno, C., Gardner, P., Zarfaty, U.: Context Logic and Tree Update. POPL(2005)
4. Gardner, P.: A Note on Context Logic DRAFT. Appsem Summer School (2005)
5. Ishtiaq, S., O'Hearn, P.: BI as an Assertion Language for Mutable Data Structures. ACM-SIGPLAN(2001)
6. Cheung, K.: High-Level XML Update: Low-Level Reasoning. MSci Project (2006)
7. Kearns, B.: Context Logic for Low-Level Update. MSc Project(2007)
8. Smith, G.: First year Report: A context logic approach to analysis and specification of XML update. (2006)
9. Calcagno, C., Gardner, P., Zarfaty, U.: Local Reasoning about Data Update. Festschrift(2005)
10. W3C: World Wide Web Consortium (2007)
http://www.w3.org/
11. W3C: DOM: Document Object Model (1998)
http://www.w3.org/DOM/
12. W3C: Document Object Model (Core) Level 1 (1997)
http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html
13. An Introduction to the W3C DOM (2006)
http://www.slayeroffice.com/articles/DOM/
14. JavaScript tutorial - DOM nodes and tree (2007)
http://www.howtocreate.co.uk/tutorials/javascript/dombasics
15. Wiklicky, H.: Applied Semantics - a 2nd Year Course in DoC at Imperial College London.
http://www.doc.ic.ac.uk/ herbert/english/e-teaching.html

10 Appendix I - Constructing Minimal DOM

10.1 Node

The full IDL definition of a Node is as follows,

```
interface Node {

  //NodeType
  const unsigned short   ELEMENT_NODE                = 1;
  const unsigned short   ATTRIBUTE_NODE              = 2;
  const unsigned short   TEXT_NODE                   = 3;
  const unsigned short   CDATA_SECTION_NODE          = 4;
  const unsigned short   ENTITY_REFERENCE_NODE       = 5;
  const unsigned short   ENTITY_NODE                 = 6;
  const unsigned short   PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short   COMMENT_NODE                = 8;
  const unsigned short   DOCUMENT_NODE               = 9;
  const unsigned short   DOCUMENT_TYPE_NODE          = 10;
  const unsigned short   DOCUMENT_FRAGMENT_NODE      = 11;
  const unsigned short   NOTATION_NODE               = 12;

readonly attribute DOMString        nodeName;
         attribute DOMString        nodeValue;
                                    // raises(DOMException) on setting
                                    // raises(DOMException) on retrieval

readonly attribute unsigned short   nodeType;
readonly attribute Node             parentNode;
readonly attribute NodeList         childNodes;
readonly attribute Node             firstChild;
readonly attribute Node             lastChild;
readonly attribute Node             previousSibling;
readonly attribute Node             nextSibling;
readonly attribute NamedNodeMap     attributes;
readonly attribute Document         ownerDocument;

    Node      insertBefore(in Node newChild, in Node refChild)
                  raises(DOMException);
    Node      replaceChild(in Node newChild, in Node oldChild)
                  raises(DOMException);
    Node      removeChild(in Node oldChild)
                  raises(DOMException);
    Node      appendChild(in Node newChild)
                  raises(DOMException);
    boolean   hasChildNodes();
    Node      cloneNode(in boolean deep);
```

};

This provides us with a list of the attributes and the methods that the Node interface provides. The behaviour of these attributes and methods is given in an English specification. For the scope of this project we are only interested in manipulating the Nodes within the tree structure of documents and not in their data content. We require only a single Node type so we omit the NodeType constants mentioned at the start of the IDL definition. For the same reason we can also omit the attributes "nodeType", "nodeValue", "attributes" and "ownerDocument". To represent whatever data is associated with a Node in a flat fashion we use a single **tag** which is of the String type.
The remaining attributes are all read only, so in the interests of language independence and model simplicity we choose to represent these attributes as a "getter method". So we have "getNodeName" for "nodeName", "getParentNode" for "parentNode" and "getChildNodes" for "childNodes". By not providing any set methods, we have in effect made these attributes read only.
Note that because "getChildNodes" returns a "NodeList" we are compelled to provide the functionality of this interface in our specification. This is detailed in the next section.
Given that we provide this "NodeList" structure and the method "getChildNodes", the attributes "firstChild", "lastChild", "previousSibling" and "nextSibling" are no longer required, as we can obtain this information from the "NodeList" implementation.
We will provide the methods "insertBefore", "removeChild" and "appendChild", but not "replaceChild", "hasChildNodes" or "cloneNode", as these commands can be simulated using the other commands we specify. (This will be proved in the content of this paper).

10.2 NodeList

The IDL definition of a NodeList is as follows,

```
interface NodeList {
  Node                              item(in unsigned long index);
  readonly attribute unsigned long  length;
};
```

This tells us that a NodeList is made up of indexed items, each of which is a Node. A NodeList also has an associated length. Notice however that we have not been given any information about how the list might be stored. It could be an array, a linked list, or any other kind of list, but this does not matter to DOM. Likewise we need to ensure that our specification maintains this platform free nature. For convenience we shall be assuming that NodeLists are 0 indexed, that is the first element is at position 0 in the list. We shall provide the method "getItem" for the specified "item" method, and we provide

the read only attribute “Length” as a getter method called “getLength”. This will keep our specification language independent and our model simple.

10.3 Constructors

Reading section 1.1.2 of the DOM Core Level 1 specification [12] we find the following statement about memory management,

> *...ordinary constructors (in the Java or C++ sense) cannot be used to create DOM objects, since the underlying objects to be constructed may have little relationship to the DOM interfaces. The conventional solution to this in object-oriented design is to define factory methods that create instances of objects that implement the various interfaces. In the DOM Level 1, objects implementing some interface “X” are created by a “createX()” method on the Document interface; this is because all DOM objects live in the context of a specific Document.*

As mentioned before, we are only interested in the structure of XML documents in DOM, and we are only working with Nodes in the scope of this project. However, the Node is an abstract concept in DOM and to construct one we would have to actually construct one of its concrete subclasses (A Text Node or Attribute Node for example). In the IDL definition above there is no mention of a “createNode” method. In any practical case, one would only be interested in the various specialised types of Node, however, due to our restriction to the structural behaviour of DOM we are interested in this generic Node type. A constructor method “createNode” for this generic Node type will be necessary for our specification, and we model this on the “createElement” method.
We do not need to provide a constructor for the NodeList interface, as a NodeList never occurs outside of the context of a Node, so they can be created together in the same constructor.

10.4 Commands

Above we briefly discussed which commands we are choosing to work with. Given below is the full list of commands that we shall be using in our specification along with a brief English description, obtained from the existing DOM specification, as to the behaviour of each command.

1. createNode(String: tag)
   creates a new node with fresh id's and its tag (representing the flat data associated with this new node) is set to the input string

2. getNodeName(Node: node)
   returns the sting stored in the tag of the node with the input identifier

3. getChildNodes(Node: parent)
   returns the identifier of the nodelist of children for the node with the input identifier

4. getParentNode(Node: child)
   returns the parent node of the node with the input identifier

5. insertBefore(Node: parent, Node: newChild, Node: ref)
   inserts the node with identifier newChild before the node with identifier ref in the NodeList associated with the node with identifier parent

6. removeChild(Node: parent, Node: child)
   removes the node with identifier child to the top grove level of the DOM trees from the NodeList of the node with identifier parent and returns the node identifier child1

7. append(Node: parent, Node: newChild)
   puts the node with identifier newChild at the end of the NodeList of the node with identifier parent

8. getLength(NodeList: fid)
   returns the length of the NodeList with identifier fid

9. getItem(Int: length, NodeList: fid)
   returns the identifier of the Node at position length of the NodeList with identifier fid

You may notice that compared to the current DOM specification some of these commands have an extra input Node. This is simply because DOM defines these commands as methods called upon Node objects and we don't want to concern ourselves with this style of argument parsing. We choose to pass the Node the method was called upon as an explicit input to that command.