NIST GCR 04-860

DOM Test Suite Methodology Report

Dimitris Dimitriadis
Ontologicon

National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

NIST GCR 04-860

DOM Test Suite Methodology Report

Prepared for
U.S. Department of Commerce
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8970

By
Dimitris Dimitriadis
Ontologicon

February 2004

U.S. Department of Commerce
Donald L. Evans, Secretary

Technology Administration
Phillip J. Bond, Under Secretary for Technology

National Institute of Standards and Technology
Arden L. Bement, Jr., Director

PREFACE

Testing tools for conformance are emerging as a key technology for enabling and maintaining interoperability between systems. However, the development of conformance test methods and tools is time consuming and labor intensive. Automated test development methods are more efficient as well as less error prone. One of the first World Wide Web Consortium (W3C) test suites to incorporate automation is the DOM Test Suites (DOM TS). This report documents the DOM TS methodology. It presents an overview of the technology and describes the goals in building the automated test generation technique, the degree to which it succeeded in meeting these goals. Additionally, it describes the needs the DOM TS was designed to fill, its architecture, and its capabilities and limitations. Finally, the possibilities to generalize the framework and some issues encountered in the Test Suite production life are discussed.

This report was written as a contract deliverable for the U.S. Department of Commerce (DoC), National Institute of Standards and Technology (NIST), Software Diagnostics and Conformance Testing Division (SDCT) in support of its standards and conformance testing program.

KEYWORDS: automated test generation, conformance testing, DOM, test case description language, test suite, W3C

Table of Contents

1. Introduction

This document serves as a description of the Document Object Model Test Suites (DOM TS), that have been successfully used to delevop tests for DOM Level1, Level2 and now Level 3. It discusses its intended goals and also describes limitations and possible extensions to the DOM Test Suites Framework. A discussion of the degree to which it follows existing W3C Quality Assurance (QA) guidelines, indicating where the DOM TS framework digresses and the reasons for that being the case, is made. Finally, an evaluation of the first two years of DOM TS life is conducted before a conclusion and pointers to future possible work is given.

The document is not technical in nature, as that aspect has been successfully covered in other documents (see [DOMTS]).

2. Overview of DOM TS Technology

2.1 Background

The Document Object Model (DOM, see [DOM]) is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of HTML and XML documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. DOM is used not only in Web environments, but can equally be used to conduct serverside data manipulation. It is therefore a good technology for testing frameworks, as it is applicable across platforms and language-neutral (thus allowing for general test guidelines to be implemented). In addition, writing test cases for DOM is relatively easy, as all DOM methods and attributes are clearly defined and delimited from one another, leaving no or little ambiguity.

The DOM specification is written using a version of XML Spec, an XML grammar used to write W3C specifications. It defines functionality or intended behavior. The DOM interfaces and methods are written using IDL (Interface Definition Language), since DOM is an interface and not a language as such. Furthermore, as an interface, DOM is implemented in particular environments by means of bindings (that is, the DOM functionality is written in different ways for different languages, keeping functionality the same across these different languages).

2.2 Reasons for designing the DOM TS

There are technical as well as non-technical reasons for writing an advanced, if possible automated, test framework.

2.2.1 Technical reasons

1. Easier to write conformant implementations if it is possible to test the implementation during development

2. Less time needed to write test cases if tests can be derived or otherwise transformed from a basic set of test case descriptions

2.2.2 Non-technical reasons

1. A good test suite makes it more attractive for implementors to write conformant software instead of implementing their own versions of the standard.
2. The W3C requires that each specification be successfully implemented by two different implementations before accepting the specification as a Recommendation.
3. Conformance claims, oftenly being used not only as technical information but also as a marketing tool, are more easily resolved if the test suite is complete and accurate.
4. Raise interest and awareness of the DOM technology.

2.3 Goal of building automatic test generation techniques

The W3C DOM Working Group (DOM WG) and NIST decided to jointly launch an activity to produce a Test Suite (TS) for the DOM specification, in order to further outreach and make it easier for DOM implementors to produce conformant software. The DOM TS group, a public forum inviting all interested parties, was formed in March 2001 in order to take on the responsibility to produce the test suite, the test running framework, a reporting and coverage mechanism, as well as write a methodology, in short, to create the W3C DOM TS (created publically and endorsed by the DOM WG).

The DOM TS was led by a DOM WG representative, forming the link between the TS group and the WG, in order to facilitate decision procedures and clarifications, in case they were needed. The DOM TS group conducted its work publically, and the DOM WG representative reported to the DOM WG in two cases:

- for resolving issues, for example differences in the interpretation of the specification in relation to the outcome of submitted tests, and
- for information relevant to releases of the DOM TS.

Concerning all other issues, the group was intended to work outside of immediate DOM WG control.

No similar work had previously been done, and for this reason many issues were in need of being solved before the actual TS work could start.

NIST had already released a DOM Test Suite independently of the W3C DOM WG. NIST donated the tests to the W3C/NIST framework, which fit in nicely with the W3C DOM WG's intentions to provide a test framework.

2.3.1 Intentions

1. Test implementation support of the DOM among browsers and DOM-enabled software (mainly XML and HTML parsers).

2. Easy and uniform testing of those implementations.
3. Make it as easy as possible to write tests in order to make it attractive for test authors and enhance the quality of the test suite. Since the DOM has two official bindings, ECMAScript and Java, it was decided from the start that ease of test authoring was a priority. Investigating this idea the group decided to allow for writing one test case and test many implementations (write a test case in a neutral language and generate the different tests for particular bindings, instead of writing multiple test cases, one for each binding).
4. Allow the framework to indicate what tests are applicable to run for those implementations that state up front the parts of the specification they support. For example, there are DOM implementations that support XML only and do not support the DOM HTML. The test framework should be able to select the appropriate tests to be run.
5. Create links to the part of the DOM specification being tested for traceability, easily resolving issues and enhance coverage reporting.
6. Create documentation of the tests.
7. Report on the success/failure of the testing process in an easy format.

2.3.2 Technical limitations

It was obvious from the start that the DOM TS was breaking new ground, since using a particular grammar to represent tests (and validate them directly using an XML schema generated from the XML version of the DOM specification itself) had not been done before. Being pioneers in the field, the DOM TS group met some initial difficulties in reaching a conclusion as to what kind of tools and software to use. The DOM TS group decided to use publically available tools to allow for all interested parties to participate and test their implementations, mainly from the Jakarta project.

Some difficulties existed because generating tests was a 4-step process (1-4 below). Each limitation is described along with the solution adopted by the DOM TS Group as well as an indication of difficulty.

Step 1: Write a test case using some "smart" markup language. This was envisioned early, but implemented with significant delay due to design issues, primarily lack of a good enough markup language.

> *Limitation**:*** The test case markup language had to be implemented, if existing, or else designed.
> *Solution:* The group decided to write tests using a similar markup to that which the DOM specification itself uses, since inventing a new language was outside the group's scope (it would amount to inventing an abstract language for an abstract language). In order to avoid unneccesary work, it was decided to stay as close to the original markup as possible, extending it for representing metadata information (such as creation date, author name, pointer to the relevant part of the specification, and so forth). This took some time and was fairly difficult since all specific information needed to be added to the markup.

Step 2: Validate these test cases using the original XML version of the DOM specification. This was brought in to the design as a means of adressing the issue of writing vaid tests. Using test representations written in XML, it was possible to validate those tests before producing the code to be run in the test framework.

> *Limitation:* It was initially not clear what validating a test for correctness amounted to.
> *Solution:* This was relatively easy, since using the particular markup allowed for straightforward validation of the test case descriptions. Test cases were written in XML, validated against the DOM specification, and only then accepted for generating code. This meant that only valid test representations made it through to the next step.

Step 3: Generate the relevant language versions of the tests (the DOM WG and TS groups decided to go for the two official DOM bindings, ECMAScript and Java) using XSLT transforms for limiting ambiguity. If the test was valid, the transform would generate valid test code.

> *Limitation:* Since the group wanted to write one test and generate several output formats, great care needed to be taken when designing the XSLT stylesheets.
> *Solution:* Keeping the DOM TS MarkupLanguage (DOM TS ML) constant, it was easy to concentrate on two things in order to be sure of the outcome:
> - write valid tests (if invalid, they would not produce code),
> - write correct XSLT transforms (to make sure the test developers needed only care about the XML test representation and not the code producing mechanism).
>
> Having valid tests on the one hand and correct XSLT transforms on the other, the group was sure that the output was correct and could be used to test DOM conformance. Also, adding more XSLT stylesheets allowed porting the test suite to other languages, multiplying the impact on the software community, without having to write new tests. This is a continuous and fairly complex task, since ensuring the correctness of the XSLT stylesheets are a key element in providing correct test instances, requiring control and rewriting.

Step 4: Provide a framework that needed to do the following:

> *Feature*: Stay close in layout to the DOM specification (levels/modules), which was a natural effect of deciding to use the DOM XML grammar to write tests.
> *Solution:* The file structure used to store tests was designed to correlate closely to the DOM specification document anatomy (the DOM specification is divided into levels and modules, the directory structure was likewise divided into hierarchies representing levels, and different folders within each hierarchy for modules). This allowed for a straightforward layout of the file structure relied on by the DOM TS and was very easy to implement.

> *Feature:* Build the test suite

Solution: An open source framework was used for building the DOM TS (a Java tool called ANT that got the test files, the DOM specifications, validated the tests and generated the code in two versions). The builds are the end product of the build tool's gathering tests, validating and transforming them, packaging them according to level and module and compressing them for download. Several builds can exist for review before the final version (for each level) gets approval from the DOM WG and is released to the public. Easy since the build tool allows adding new specific tasks (for example, download the specifications, validate the tests, generate the schema, generate code etc.) that can be combined to produce a downloadable, ready to run test suite.

*Feature**:*** Run the tests
Solution: Two open source solutions were used to run the tests, JsUnit, a web-browser driven JavaScript engine to run tests and report results for the ECMAscript tests, and JUnit, a similar solution for the Java environment. This was difficult since some user agents did not support the test running framework as it was initially written, which meant both writing particular mechanisms for the test suite, but also interacting with the tool author to enhance the tool in subsequent releases (which is a good showcase of interaction between test suite authors and open source tool providers).

Feature: Report on implementation's success or failure.
Solution: This was easily done in each tool separately as they both contain mechanisms for success/failure.

Feature: Generate on overview of results.
Solution: Done using parsing of results. Ant, the tool used to build the test suite, is used for this purpose as well.

Feature: Provide links to the test cases themselves.
Solution: Links were provided to the test storage space and to the local checked out version of the test suite. These tests could be browsed from a table built by the building tool to allow for test case authors to look at the actual test. Very easy since it amounts to pointing to files in a file system (local or remote).

Feature: Provide links to the part of the specification being tested directly from the test to add traceability.
Solution: Easy, since each test contains a pointer to the part of the specification where the DOM functionality to be tested is specified. This way, test case authors can look at the part of the specification a particular test is written for, or, if writing a new test, provides a mechanism to point to the relevant aspect of the specification. Very easy since this information is present in the test case description.

Feature: Be able to decide what subset of the test suite to run, given implementation characteristics. For example, some implementations support only

the XML, and not the HTML, DOM. Therefore, tests need to be marked according to what DOM module they were written for, in order to allow for implementations that do not support that module to run only relevant tests and not have reports of test failures.
Solution: Also fairly easy, since the DOM specifies, as part of its functionality, an indicator of what the implementation supports. The implementation could be asked what it supports before the relevant tests were run.

Feature: Make it easy to submit tests.
Solution: A dedicated mailing list was set up for developers to submit tests (in the DOM TS ML language) for review. Once accepted, the test was added to the file management system and was included in the build process to form part of the test suite. Very easy and part of the first design issues.

2.4 Needs DOM TS was designed to fill

Except for the intentions described above, the DOM WG had set up a number of things that needed to be adressed in designing the DOM TS. In particular, the interoperability and language neutrality of the DOM itself should be a feature of the DOM TS as well. Designing a framework that is difficult to use is counterproductive. Also, it should be easy to construct test material for other bindings than the two official ones, thus not ruling out efforts to write test material for other than those bindings (which is a great example of DOM interoperability). In addition, the framework for executing the test suite should not need to rely on a particular technology, especially in the HTML/web case, using a browser with JavaScript capabilities should be enough. Lastly, it should be made easy for implementors to build and use the test suite, locally or remotely.

2.4.1 Multi-language support

Support for several languages was achieved, as indicated above, by using one primary language to write the test cases (the DOM TS ML), and then transform the test to each of the desired bindings. This way, all that needed to be done to transform the bulk of tests to a new binding was to provide another XSLT stylesheet which would be invoked in the build process to generate the tests for other than the two original bindings.
This was important since duplicate work, especially rewriting test cases for all bindings separately, was ruled out. Also, it was intended as a show case of the DOM interoperability design.

2.4.2 Simplistic execution framework (HTML/web case)

In order to be able to test as many user agents as possible, the group decided to rely on an architecture that required support for the so called DOM Level 0, or dynamic HTML. This level does not exist as a specification, but served conceptually as the predecessor to DOM level 1. This technology is supported by most web browsers. What was not done was to allow for platform specific features to form part of the DOM TS, for example file loading mecahnisms present in some browsers but not others.

To accomodate for all differences, the DOM TS Group decided to use the common denominator as the core for the DOM TS framework. This was an HTML 4 compliant browser with support for ECMAScript.

2.4.3 Easy to build and execute

The DOM TS should be accessible to implementors using as few proprietary technologies as possible. Implementors should even be able to build the test suite remotely, allowing for testing implementations on platforms that the build platform itself does not support. Building is thus separated from executing and testing.
It should also be possible to run the test suite over the internet, allowing for the test suite to exist at a location physically different from that of the implemention being tested.

2.5 DOM TS architecture

2.5.1 Interoduction

Based on the concepts described in the previous section, the DOM TS was designed to meet those goals using several mechanisms. The DOM TS is comprised of a number of components, each of which will be introduced below.

2.5.2 Test cases and Test Case Description Language

The DOM TS starts from the assumption that tests can be described independently from the test instances (the actual code used to test comformance). This is done using a meta language which is used to write the test cases used to generate particular test instances for each desired binding. Furthermore, DOM TS uses tests that have been checked for correctness, or validated, against a schema which, in turn, is generated from the XML version of the DOM specification itself. The specification is written using a particular XML grammar (a version of XMLSpec, fairly widely used in W3C specification authoring), and from this XML version the normative published HTML version is generated. The meta language is generated from the DOM XML Spec using XSLT stylesheets. Both DTD and W3C Schema versions of the DOM TS ML (DOM TS MarkupLanguage) schema are generated and used when test suites are built since all test are validated before they are used to create test instances.

A test case description language (TCDL) is constructed using XSLT stylesheets to generate the DOM TS ML (which is the TCDL for DOM) from the XML version of the DOM specification. The DOMTSML build on concepts in the DOM specification and is extended with constructs for metadata and programmatical information, such as variable declaration.

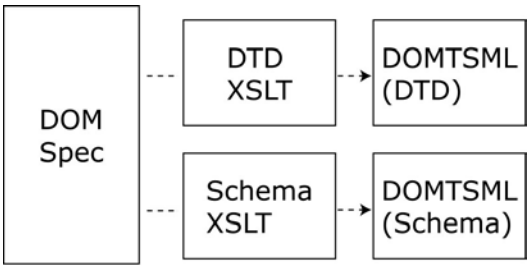

Figure 1: TCDL construction

Tests are written using this meta language and are validated against the XML version of the DOM specification.

Example of the code generation technique

The XML document below is the test description for getting the value of an attribute node in an external XML document. This is stated in the test, both in prose (the test purpose), which is copied to the test table, and in the DOM TS ML language in the test itself for including in the actual test instance. The test itself is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Copyright (c) 2001 World Wide Web Consortium,
(Massachusetts Institute of Technology, Institut National de
Recherche en Informatique et en Automatique, Keio University). All
Rights Reserved. This program is distributed under the W3C's Software
Intellectual Property License. This program is distributed in the
hope that it will be useful, but WITHOUT ANY WARRANTY; without even
the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
See W3C License http://www.w3.org/Consortium/Legal/ for more details.
--><!DOCTYPE test SYSTEM "dom1.dtd">

<test xmlns="http://www.w3.org/2001/DOM-Test-Suite/Level-1"
   name="attrname">
<metadata>
<title>attrName</title>
<creator>NIST</creator>
<description>
  the getNodeName() method of an Attribute node.
  Retrieve the attribute named street from the last
  child of of the second employee and examine its
  NodeName.  This test uses the getNamedItem(name)method
  from NamedNodeMap interface.
</description>
<contributor>Mary Brady</contributor>
<date qualifier="created">2001-08-17</date>
<!--   Node.nodeName  -->
<subject resource="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
core#ID-F68D095"/>
<!--   Attr.name      -->
<subject resource="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
core#ID-1112119403"/>
</metadata>
<var name="doc" type="Document"/>
<var name="addressList" type="NodeList"/>
<var name="testNode" type="Node"/>
<var name="attributes" type="NamedNodeMap"/>
<var name="streetAttr" type="Attr"/>
<var name="name" type="DOMString"/>
<load var="doc" href="staff" willBeModified="false"/>
<getElementsByTagName interface="Document" obj="doc" var="addressList"
  tagname="&amp;quot;address&amp;quot;"/>
<item interface="NodeList" obj="addressList" var="testNode" index="1"/>
```

```
<attributes obj="testNode" var="attributes"/>
<getNamedItem obj="attributes" var="streetAttr"
  name="&quot;street&quot;"/>
<nodeName obj="streetAttr" var="name"/>
<assertEquals actual="name" expected="&quot;street&quot;" id="nodeName"
  ignoreCase="false"/>
<name obj="streetAttr" var="name" interface="Attr"/>
<assertEquals actual="name" expected="&quot;street&quot;" id="name"
  ignoreCase="false"/>
</test>
```

In the build process, the test is validated against the DOM specification itself to make sure it is correct and there are no ambiguitites. The validation is done against a DTD or Schema file which itself is generated directly from the DOM specification, again using a stylesheet.

2.5.3 Test generation and adaption to test execution frameworks

Tests are transformed into the two official DOM language bindings (ECMAScript and Java) using XSLT stylesheets for transforming the test representations into executable code. The code is in turn run in two frameworks, JSUnit for ECMAScript, JUnit for Java.

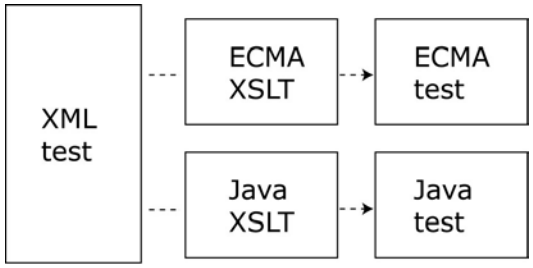

Figure 2: generating the code

The valid tests are transformed into each of two bindings (more can be added), ECMAScript and Java, in the appropriate form for being run into the two frameworks.

Applying the appropriate XSLT stylesheet, the output is ready to be plugged into the JsUnit framework and generate a result. The script in the HTML page that JsUnit uses looks as follows:

```
/**
*
    The getNodeName() method of an Attribute node.
  Retrieve the attribute named street from the last
  child of of the second employee and examine its
  NodeName.  This test uses the getNamedItem(name) method from the NamedNodeMap
  interface.

* @author NIST
* @author Mary Brady
* @see <a href="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
core#ID-F68D095">
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core#ID-F68D095</a>
* @see <a href="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
core#ID-1112119403">
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core#ID-
1112119403</a>
*/
function attrname() {
    checkSetUp();
```

```
    var doc;
      var addressList;
      var testNode;
      var attributes;
      var streetAttr;
      var name;
      doc = load(this.doc, "doc", "staff");
      addressList = doc.getElementsByTagName("address");
      testNode = addressList.item(1);
      attributes = testNode.attributes;

      streetAttr = attributes.getNamedItem("street");
      name = streetAttr.nodeName;

      assertEquals("nodeName","street",name);
       name = streetAttr.name;

      assertEquals("name","street",name);

}

Applying a different stylesheet, the Java code is generated:
/*
This Java source file was generated by test-to-java.xsl
and is a derived work from the source document.
The source document contained the following notice:


Copyright (c) 2001 World Wide Web Consortium,
(Massachusetts Institute of Technology, Institut National de
Recherche en Informatique et en Automatique, Keio University). All
Rights Reserved. This program is distributed under the W3C's Software
Intellectual Property License. This program is distributed in the
hope that it will be useful, but WITHOUT ANY WARRANTY; without even
the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
See W3C License http://www.w3.org/Consortium/Legal/ for more details.

*/

package org.w3c.domts.level1.core;

import org.w3c.dom.*;

import org.w3c.dom.events.*;

import org.w3c.domts.DOMTestCase;
import org.w3c.domts.DOMTestDocumentBuilderFactory;


/**
*
    The getNodeName() method of an Attribute node.
  Retrieve the attribute named street from the last
  child of of the second employee and examine its
  NodeName.  This test uses the getNamedItem(name) method from the NamedNodeMap
  interface.

* @author NIST
* @author Mary Brady
* @see <a href="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
core#ID-F68D095">
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core#ID-F68D095</a>
* @see <a href="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
core#ID-1112119403">
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core#ID-
1112119403</a>
*/
public class attrname extends DOMTestCase {

   /**
    * Constructor
```

```
     * @param factory document factory, may not be null
     */
    public attrname(DOMTestDocumentBuilderFactory factory)  {
       super(factory);

    }

    /**
     * Test body
     * @throws Throwable Any uncaught exception causes test to fail
     */
    public void runTest() throws Throwable {
       Document doc;
       NodeList addressList;
       Node testNode;
       NamedNodeMap attributes;
       Attr streetAttr;
       String name;
       doc = (Document) load("staff", false);
       addressList = doc.getElementsByTagName("address");
       testNode = addressList.item(1);
       attributes = testNode.getAttributes();
       streetAttr = (Attr) attributes.getNamedItem("street");
       name = streetAttr.getNodeName();
       assertEquals("nodeName", "street", name);
       name = streetAttr.getName();
       assertEquals("name", "street", name);

    }
    /**
     *  Gets URI that identifies the test
     *  @return uri identifier of test
     */
    public String getTargetURI() {
       return "http://www.w3.org/2001/DOM-Test-Suite/level1/core/attrname";
    }
    /**
     * Runs individual test
     * @param args command line arguments
     */
    public static void main(String[] args) {
         DOMTestCase.doMain(attrname.class, args);
    }
}
```

2.5.4 Test case table and traceability

In addition, XSLT stylesheets are used to produce a table which contains a pointer to each test, part of specification tested by the test, any prose provided in the test description (the test purpose in plain English) as well as a pointer to both the JavaScript and Java version of the test (stored locally). This table can be used to evaluate the coverage of the specification in terms of tests written for its various modules/aspects and provides traceability from the test to the relevant part of the specification.

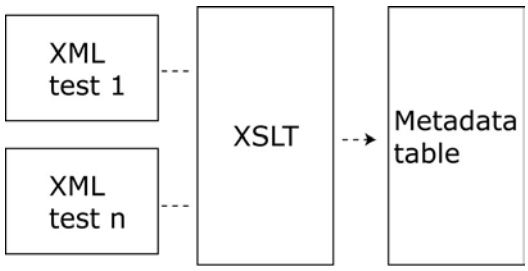

Figure 3: generating the metadata table

A separate XSLT stylesheet generated a table which contains the test case name, id, the test case description, a pointer to the specification, and the XML version of the test itself (both language versions).

In the test case above, there are links to the relevant part of the DOM specification being tested which are used in the table. The indications are:

```
<!--   Node.nodeName  -->
<subject resource="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
  core#ID-F68D095"/>
<!--   Attr.name      -->
<subject resource="http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-
  core#ID-1112119403"/>
```

that point to the exact wording of the definition of the function that is being tested. This gives a direct means of controlling test success.

2.5.5 Data management

For simplicity, a Concurrent Versioning System (CVS) is used for test case versioning and storage (see [DOMTSCVS]). The tests are stored in a file hierarchy correlating to the DOM specification document anatomy. The specification is written in levels, each consisting of modules (Level 1 is divided into Core and HTML, Level 2 into Core, HTML, Events, Style, Traversal and Range and Views). This layout is used by the file management system as well and is represented by an equivalent folder structure. This allows for using outdated builds (if implementors need to check their implementation against any given version of the test suite they can). It also allows for easy resolvment of issues, since there is a bug tracking system into which errors are reported and discussed in the DOM TS group and, if needed, by the DOM WG. Once resolved, the correct test case gets uploaded to the CVS tree, is marked appropriately and is used from then on. When the test case development is frozen, a downloadable version of the DOM TS is released and bears the DOM WG "official" stamp. Up until that point, any implementor can create test suites using the tools that the DOM TS relies on to test their implementation for correctness.

The CVS contains all files necessary to the DOM TS. Except for the tests itself, it contains the relevant stylesheets, submitted tests pending approval, documentation and the script describing the build process.

2.5.6 Overview

The DOM TS comprises of several parts, that are interconnected as shown in Figure 4 below.

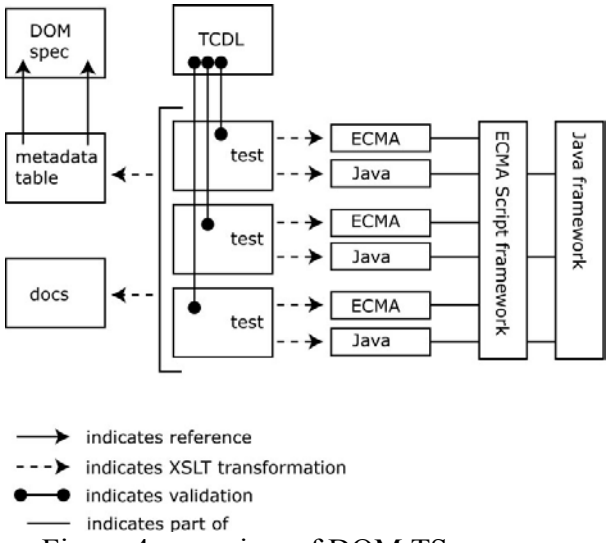

Figure 4: overview of DOM TS

1. The XML version of the DOM Specification is used to generate the DOMTSML.
2. The TCDL is used to write the actual test descriptions. The build process stipulates that only valid tests, that is, tests that have successfully been checked for correctness, can be used to generate code.
3. The test execution frameworks use only valid tests derived from the test descriptions. Only valid tests are allowed to form the raw material for executable code and as a consequence, only valid tests (that is, tests written in the DOMTSML) need to be written in order to test for conformance.
4. Furthermore, documentation is generated from the set of valid tests (the set of tests that make up the test suite).

2.6 DOM TS methodology

The DOM TS methodology is very simple. This is due to the fact that the DOM TS was launched and designed at a time when there were no generally accepted guidelines to follow, in particlar no W3C endorsed Quality Assurance guidelines as the W3C QA WG had not yet been formed. Most of the DOM TS was designed from scratch rather than rework existing methodologies, for example, the choice of tools necessary to build the DOM TS and the choice of frameworks used to run the DOM TS. One of the objectives was to use publically available open source tools, in order to reuse and not reinvent existing solutions (for example, CVS provides a framework for versioning control that was easy to include in the DOM TS architecture). Had the DOM TS been designed with guidelines existing today in mind, it may have turned out slightly different in some respects. Its main characteristics would have remained the same, however.

The strengths of the DOM TS methodology are its platform independence and independent development from the DOM WG. The platform independence of the ECMAScript version of the DOM TS ensures that any browser supporting basic DOM (also called DOM Level 0) can run the DOM test suite. The methodology was developed by a task force (the DOM TS Group) separate from the DOM WG, thus providing an independent look and interpretation of the specification upon which the test suite is built. Interaction between the DOM TS Group and DOM WG was limited, constisting of

guidance as to current releases and normative interpretations of the specification. Thus the DOM TS Group was able to focus on providing a high quality test suite. This approach proved to have strengths in that the groups was focused and not influenced by "what was meant in the specification" but rather what the specification actually says. The weakness of this approach is that there were delays in receiving normative interpretations (on specification issues) and decisions (for example on releases) from the DOM WG and, consequently, making decisions based DOM WG interpretation.

The main characteristic of the DOM TS is that a test is not understood to be an executable set of code; rather is is understood as a meta level representation of DOM behaviour, viewed as an atom, which in turn is used to

1. generate an actual test in executable code format for two bindings (ECMAScript and Java)
2. form part of a test suite, which is a number of collections of test cases, grouped similarly to how the DOM modules are architected. It is easy to construct different test suites for XML and/or HTML, using the same test cases for both in some instances. A test suite is understood as a collection of tests, grouped in accordance with the modules/parts of the specification.

Furthermore, the difference between test case and test representation allows for easier test case management as the latter, and not the former, serve as raw material for the actual test suite. A test representation (a test) is a description of intended behaviour, which, since it is is written in rich markup, is used to generate actual test cases, that have in turn been validated for correctness and are only then used to control conformance. Any metadata, such as version, part of the specification, and so on, is set on the test representation level, allowing for the build tools to configure, based on this metadata, the appropriate sets of tests (test suites) corresponding to the various DOM spefication levels. This does not hinder one test being used for more than one suites, but is required as data in the build script. The reason for using this approach is that the test can be written in a fairly simple and agnostic manner, as it serves as raw data and need not contain much more information than the most fundamental. Tests need not contain information on coverage, part of specification, and so forth, since this can be regulated on the test suite level (using the build tool). This can in turn be physically different from the test storage, allowing for many different means of conglomerating tests into test suites.

The logical distinction between test and (test) suite is important since the former represents testability aspects (on machine level, executable code is generated from these test) and the latter represents coverage and reporting issues (grouping, successes/failures and so forth).

2.6.1 Strengths – advantages

The methodological advantages of the DOM TS have been discussed in previous sections. Here, a list of the DOM TS strengths from a technical perspective is given.

2.6.1.1 Tests for multiple languages

The DOM TS allows for easy creation of tests for multiple languages in one test suite; while supporting the two official bindings (ECMAScript and Java), other languages (such as Python, C++ etc.) are possible to generate as well. This furthers the DOM aims of interoperability. It also simplifies the task of the test developer, as they need only write one test, which in turn is transformed into the intended language binding.

2.6.1.2 Specification area coverage

Since the tests are written in an XML grammar similar to that which is used by the DOM specification itself, it is fairly easy to programmatically control what aspects of the specification have corresponding tests. Using web technologies, one can check, for example, that all DOM methods and attributes have corresponding tests. This way, a simplistic coverage map can be provided, allowing for the test author to concentrate on areas that there are no, or few, tests for.

2.6.2 Limitations – requirements

2.6.2.1 Using markup to write tests

Given the decision to use a rich markup for describing behaviour and writing tests, the technical threshold was initially quite high. This was overcome as soon as the added value introduced by the write one test case, generate many tests was seen. For some, non-technically oriented people, however, it seemed more straightforward to only write executable code and package it to produce a test suite than to write a test descritpion, validate it, produce executable code, have it packaged and then executed in the form of a test suite.

2.6.2.2 Using specifications that have been written using an XML grammar

The DOM TS requires XMLSpec-based specifications (or specifications using similar functionally rich markup language) in order to allow for test generation. Tests written for a specification using plain HTML are not suitable for automated processing, as there is no markup to use to create the test language nor any rich markup serving as the basis of the specification; thus nothing to validate against. It is also not equally easy to point to the relevant part of the specification being tested, except if some fairly advanced id/anchoring mechanism has been used to identify relevant parts of the specification. If more specifications were written using rich markup, the DOM TS could serve as a general proposal for test suite production.

2.6.2.3 Suitable for interface-style specifications

The DOM TS and similar frameworks work best for interface or function style specifications (DOM, XSLT and the like) and not equally well for prose-based, explicitly user-oriented specifications (WAI, for instance), as these are not supported by an agreed on markup language which would be used to describe the specification's intended behaviour. This in turn limits all functionality pertaining to test case description, pointing to test part of the specification, specification area coverage reporting, and the like.

3. Extending/generalizing DOM TS methodology

3.1 Extensions

A similar design could be used for

- Any specification that is written using a semantically rich markup language (with clear indications of expected behaviour provided). Some examples are XML, XSLT, XPath and XML Schema.
- Testing implementations that span several specifications, as interdependancies can be modelled (again, markup is the limiting factor). This approach presupposes that clear distinctions between specifications are made and can be represented in the markup language. One example of this is DOM Level 1 requiring support for HTML 4.
- If a granular grammar is used to write the specification, generation of basic coverage tests is made relatively easy, by letting an automated tool generate, for example, one basic functionality test per behaviour/method. This can then be used as a primer to build a more complete test suite. Exposing all relevant functionality allows for programmatical generation of tests for each function.
- Let the build tool gather all results and present them in the coverage table. Coverage can also be automated, if the script asks the specification what functions it has and checks to see if there are valid tests for all functions.

3.2 Generalizations

Same type of test generation technique (or, at least, test validation means) could be used for other technologies. Again, kind of technology specified places limitations, as some specifications do not lend themselves easily to this kind of approach (WAI, other specifications of technologies that are not "machine-aware", like DOM).

Generating tables for results reporting could be made in similar ways for other test suites (test id, result [pass/fail], link to relevant part of specification, source code).

4. Areas where DOM TS methodology is suitable/unsuitable

The DOM TS is, as mentioned above, suitable for interface-style specifications, or any specification which is either written using rich markup (allowing for writing tests at a meta level), or specifications that are inherently technical in character.

It is not equally suitable for, say, human interaction specifications, or accessibility guidelines. It may be well suited for those cases where the specification describes what the intended behaviour of an implementation claiming to conform to such and such a level of it is, for example behaviour of a user agent when presenting items in a drop down list, or highlighting text areas on mouse over.

Any specification which is discreet and quantifiable can serve as a basis for a DOM TS style approach. Specifications that are less clearly quantifiable and specify qualitative behaviour are not equally easy to produce such frameworks for.

These remarks apply mainly to the test case representation and automated test generation part of the DOM TS. For other, equally important parts, such as the documentation, coverage table or test running framework, the remarks do not apply. For any kind of specification, it is wise to use a streamlined process to produce, evaluate and display reports of the running parts of the framework.

As far as licensing issues are concerned, using a design that is released under a particular license is not a good idea, especially not if the implementation to be tested clearly states that it cannot be used with data released under such and such a license. This indeed happened with the DOM TS and limited its impact.

5. General remarks

In the DOM TS case, some companies ran into licensing issues, not allowing them to contribute to the test suite, and even more important, to run it on corporate machinery (as the test suite was released under an uneccaptable, to them, license). This obviously means that the impact any such test suite can achieve is seriously limited. It also means that those companies would under no circumstances accept claims made by others that have run the test suite for them regarding their implementation's compliance or non-compliance to a specification. Had the test suite been designed in the original chartering time space, this issue would surely have been adressed, as any company could have decided on participating in developing the test suite, or at least adressing the licensing issue before the design started. It is of course vital to construct test suites that are runnable on as many platforms as possible, otherwise the whole endeavour is seriously hindered.

Lack of incentives for submitting tests to the framework as well as allocating resources meant that "big actors" did not participate in the work, thus reducing the impact the test suite could have on the community in addition to making the work fairly slow in progress. In the DOM TS case this refers to the member companies as well as other major companies that showed no, or late, interest. It is therefore very positive to see that current W3C QA work produces tools and guidelines that aim at solving these issues before any design starts. In addition, each company can, early in the life of a WG, decide on how to position itself with regard to the design and results of the test suite.

Having no back up from large companies means there is less chance of the test suite being generally adopted by the community, regardless of whether this is because of licensing issues or resource allocation. Finally, outreach is, in effect, minimized. This is surely a bigger problem than technical limitations and design issues and belongs to a grey zone between methodology and commitment to producing test materials.

6. References

Glossary

binding: functionality in a particular environment or language

build: the end product of a programmatic process generating tests and grouping them under certain criteria to form a test suite or generate documentation

DOM: a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of HTML and XML documents.
DOM TS: a test suite for the DOM specification.
DOM TS ML: the markup language used to describe test cases used in the DOM TS
framework: software and programs used to execute the tests and gather results
IDL: Interface Definition Language, an abstract language to describe intended behaviour and functionality of programs
test case: a description of intended behaviour
test case description language: language used to describe a test case
test instance: a test in a particular language
test suite: a collection of tests of an implementation to ascertain its conformance to a specification

Documentation

DOM main page

[DOM] http://www.w3.org/DOM

DOM Test Suites main page

[DOMTS] http://www.w3.org/DOM/Test

DOM TS CVS

[DOMTSCVS] http://dev.w3.org/cvsweb/2001/DOM-Test-Suite/

Software used

Ant: http://ant.apache.org/

JUnit: http://junit.sourceforge.net/

JsUnit: http://www.jsunit.net/