Applying Model-based Testing to HTML Rendering Engines – A Case Study*

Jens R. Calamé[1] and Jaco van de Pol[2]

[1] Centrum voor Wiskunde en Informatica,
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
[2] University of Twente, Faculty EEMCS
P.O.Box 217, 7500 AE Enschede, The Netherlands
jens.calame@cwi.nl,vdpol@cs.utwente.nl

Abstract. Conformance testing is a widely used approach to validate a system correct w.r.t. its specification. This approach is mainly used for behavior-oriented systems. BAiT (Behavior Adaptation in Testing) is a conformance testing approach for data-intensive reactive systems. In this paper, we validate the applicability of BAiT to systems, which are not behavior-oriented (reactive) but document-centered.
In particular, we apply BAiT to the test of the HTML rendering engine Gecko, which is used by Mozilla Firefox. In order to do so, we formally specify a part of the CSS box model in the specification language μCRL and implement a wrapper for the Gecko renderer. Then, we automatically generate test cases and run tests with BAiT in a controlled experiment in order to demonstrate our approach on the relevant part of Gecko.

1 Introduction

Testing as a dynamic approach to software quality assurance is widely accepted in industry and is a well-studied field in academia. State-of-the-art testing approaches are model-based [4], i.e. tests are not generated in an ad-hoc manner, but founded on the specification of the software product under consideration. One of the most rigorous model-based test approaches is conformance testing, which tests whether an implementation conforms its specification focusing on functional requirements. These notions are made precise in the formal methods community [12].

In this paper, we want to apply conformance testing to rendering engines of web browsers. In state-of-the-art webdesign, content and design are kept separate from each other. Content is defined in the Hypertext Markup Language (HTML), while the design is specified in a Cascading Style Sheet (CSS). When a web page is rendered, the information from the CSS is used to position elements of content on the rendered page. If a web document has a complex structure, rendering algorithms can turn out to be erroneous, leading to "broken" web pages with mispositioned elements. Rendering a modern web application, whose

* Part of this research has been funded by the Dutch BSIK/BRICKS project.

appearance is dynamically changed on the client side using script languages, like web applications based on *Asynchronous JavaScript and XML* (AJAX, [16]), is even more demanding for rendering engines. Performing a sufficient conformance test in this context is tedious, so that an automated solution is preferable. In this paper, we present a feasibility study for automated testing of rendering engines using the test tool BAiT.

Here, we validate the applicability of BAiT [5, 6], a blackbox test execution tool for non-deterministic, data-oriented reactive systems, to test the rendering engine of a web browser w.r.t. the positioning of boxes in the CSS box model. Boxes in HTML are entities like for instance a complete HTML document (`body` element) or a paragraph (elements `p` or `div`), which contain content or other boxes and which are positioned either absolutely or relative to each other. The box model is part of the W3C CSS Specification [15, Sect. 8].

We present a feasibility study, applying conformance testing using BAiT to the rendering engine *Gecko*[3], which is used by the open source web browser *Mozilla Firefox*[4]. In order to perform the tests, we formalize the CSS specification and design test purposes. Furthermore, we implement a wrapper component between the tester and *Gecko* in order to achieve a mapping between an action-oriented specification and the document-oriented rendering engine. For several setups of web pages, we then automatically generate parameterizable test cases. Those test cases can be instantiated with varying data settings for the positions of boxes, so that they are reusable for different page layouts. Then, the test cases are executed automatically against the test wrapper and the results retrieved from *Gecko* are interpreted in order to automatically assign verdicts.

1.1 Related Work

There exists a number of static test suites for the rendering capabilities of web browsers. Each of those test suites consists of a set of HTML and CSS documents with different page layouts. The most well-known one is probably the *ACID 2 Test*[5]. It tests web browsers for their full compliance to the actual version of CSS by rendering a web page with a vast amount of CSS features enabled.

Another set of test suites for the standard compliance of web browsers are the *W3C Cascading Style Sheets Test Suites*[6]. Here, again, we have a set of static documents, which test rendering capabilities for distinct features of CSS. Finally, Mozilla Firefox itself provides a set of static layout regression tests[7], which can be run in debug builds of the software.

Most of the named test suites are, however, not automated. In fact, the files in the test suites, i.e. the test cases, have to be loaded into the browser and then the result of rendering the page has to be visually assessed. This process is not

[3] `http://www.mozilla.org/newlayout/`
[4] `http://www.mozilla.com/en-US/firefox`
[5] `http://acid2.acidtests.org/`
[6] `http://www.w3.org/Style/CSS/Test/`
[7] `http://www.mozilla.org/newlayout/doc/regression_tests.html`

automatic at all, neither on the level of test case generation, nor on that of test execution. This means, that a certain amount of test cases has to be designed and executed manually and the results have to be visually evaluated. This process is time-consuming compared to an automated test process, where test cases – or at least test data – is generated for a number of standard and critical cases. In this case, the number of test cases to be generated and executed can be optimized in order to reduce the absolute number of test cases. The regression tests of Mozilla Firefox are at least automated on the level of test execution, however, they are still founded on a static set of test cases.

The approach, which we propose in this paper, provides not only an automated test execution and evaluation of rendering results for a fragment of the CSS features, the box model, but also an automatic generation and variation of tested web page layouts. We chose this fragment, because rendering results, i.e. the position of a box, can be objectively measured (in pixels) rather than having to be visually assessed. The approach of a fully automatic test case generation and execution has the advantage regarding the other named approaches, that the executed test cases can cover more variation w.r.t. data parameters (i.e. the position of boxes), but also regarding the structure of the rendered web pages. The first issue enables us to reuse equally structured test cases (i.e. web pages) and by that to reduce the number of generated test cases. The latter allows us to test rendering web pages with a different interrelation of elements and by that cover a larger variety of possible failures in the IUT.

We are aware of several case study reports of model-based testing, concerning topics like the Conference Protocol [2], the Storm Surge Barrier in the Netherlands [9], smart card applications [8], the telecommunication sector [3] or – vertical to our work – the generation of test purposes for the Session Initiation Protocol [1]. To the best of our knowledge, however, this is the first application of model-based automatic test generation and testing techniques to document-centered applications, esp. to HTML rendering engines.

2 The Test Environment

The test environment for our case study consists of two main components. On the one hand, we have the tester, which controls the run of the experiment. The tester will be discussed in Sect. 2.1. On the other hand, we have the implementation under test (IUT), which we will discuss in Sect. 2.3. This is the object under consideration, which we will actually be testing throughout the case study. In Sect. 2.2, we will give an introduction to the CSS box model.

2.1 Firefox and Cascading Style Sheets

Firefox. Mozilla Firefox is a stand-alone web browser, which has its roots in the Netscape Communicator from the 1990s. Most of its code was put under an open-source license in 1998 and founded the basis for the Mozilla Suite, from which Firefox arose as a stand-alone browser in 2004.

Web page content (HTML):

```
...
<div class="warning">
  ...
  <div class="warning"
       id="warning1">...</div>
  ...
</div>
...
```

Web page layout (CSS):

```
div.warning {
  border-style: solid;
  border-color: red;
  color:red;
}
div.warning#warning1 {
  font-style: italic;
}
```

Fig. 1. Two differently formatted boxes

A web browser reads and interprets HTML structured data to display web pages. The task of actually displaying is carried out by a rendering component. While loading a web page, this component incrementally builds up a Document Object Model (DOM) tree from the HTML document to be displayed together with declarative layout information. Mozilla Firefox uses the renderer *Mozilla Gecko*, whose version 1.8.1 we consider in this paper as the IUT.

Cascading Style Sheets (CSS). In the 1990s, a declarative stylesheet language was developed for structured documents in order to properly divide the content of a web page from its design. Currently, the de-facto standard for CSS is in version 2.1 [15]. This version is currently not fully supported by all web browsers, including Mozilla Firefox. This issue, however, does not affect our case study.

The CSS design definition for a web page can be provided in three different ways: as an external CSS file, which is linked to the HTML file of the web page, inline the HTML web page and inline a particular element of the web page. In the first two cases, a stylesheet is a collection of blocks of the following form:

```
element.class#id {property_1: value_1; ...; property_n: value_n; }
```

The literal `element` denotes one of the possible elements of HTML [14], like – for simple boxes – `div` or `span`. The literal `class` denotes a user-defined specialization of an HTML element, while `id` denotes a user-specific identifier for a particular occurrence of this element in an HTML document. For each of these elements, we can define pairs of properties and values.

Fig. 1 shows a small example: Boxes of class *warning* are rendered with a red border and red text. The one warning box with the identifier *warning1* additionally has the text in *italic*. Since this box is a warning box, too, it also takes over all properties from the warning boxes (red border and red text). As a result, the shown HTML code fragment is rendered as two red boxes, embedded into each other, of which the inner box has italic text.

If a CSS design definition is provided in a separate file, it is linked to the web page by using the `link` element of HTML in the following way:

```
<link rel="stylesheet" type="text/css" href="mycss.css" />
```

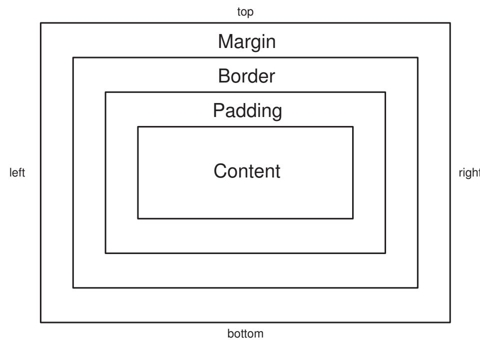

Fig. 2. CSS Box Model [15], box dimensions

CSS information can also be provided inline the HTML document by nesting it in a `style` element. Inline CSS on HTML element level, which we will be using in our case study, omits the block structure from Fig. 1. The definition of the outer box from the figure using CSS inline the element itself looks as follows:

```
<div style="border-style:solid;border-color:red;color:red;"></div>
```

2.2 The CSS Box Model

In our case study, we will regard the positioning of `div`-boxes by the Gecko rendering engine. Therefore, we have to regard both the dimensions of a box as well as other parameters, which determine the box's position or its distance to other elements on a web page. The interrelation of boxes on a web page is defined by the CSS box model [15, Sect. 8], which we will briefly introduce here.

The dimensions of a `div`-box are essentially determined by two CSS properties: `width` and `height`. Furthermore, a minimum width and height can be defined as well as their maximum counterparts.

In addition to its width and height, a box also has a number of distances to contained or surrounding content. Those settings are displayed in Fig. 2. First of all, this is the distance to surrounding content (CSS properties `margin` or `left-/top-/right-/bottom-margin`, resp.). Furthermore, the distance of the box's border to any contained content can be defined (properties `padding` or `left-/top-/right-/bottom-padding`, resp.). Finally, the width of a box's border is defined by the properties `border` or `left-/top-/right-/bottom-border`, resp. We will later come back to these settings.

Boxes can be positioned in a variety of possibilities. The positioning mode is set in the CSS property `position`, which can have one of the four values `static`, `absolute`, `fixed` and `relative`. The default setting is `static`, which does not affect the standard element flow (top to bottom on the web page). Boxes can furthermore be positioned absolutely to either the HTML document under consideration (`absolute`) or the viewport, i.e. the browser window or a

page in print (`fixed`). Finally, boxes can be positioned relative to each other, using the setting `relative`.

An absolutely positioned box is provided with up to four additional parameters determining its position: a `left`, `right`, `top` and `bottom` offset. A box, which is positioned w.r.t. the upper left corner of the HTML document can, for instance, be determined by a `left` and a `top` offset in addition to its `width` and `height`. A box with the lower right corner as its fix point would accordingly be defined using the `bottom` and `right` offset parameters and leaving the other ones undefined. For boxes, which are overdefined, e.g. by defining a `left` and a `right` offset as well as a `width`, the W3C documents define the correct handling.

While the position of an absolutely positioned box is only determined by the given offsets, the position of a relatively positioned box must be computed regarding the other boxes on the same web page. One issue which determines the position of a relatively positioned `div`-box w.r.t. another one is its position in the DOM-tree of the HTML document. If a box A appears in the tree before another box B, then A is rendered above or left of B. If A appears after B, then it is rendered either right of B or below B. Furthermore, A can enclose B, if B is a child node of A in the DOM tree.

The absolute position of the box is then computed as the summation of the other boxes' measurements. Assume, the box under consideration is anchored to the upper left corner of the web page. Then, its top offset is the sum of all top and bottom margins, widths of the top and bottom borders and the heights of all boxes *above* the one under consideration. The box's left offset is computed as the sum of all left and right margins, widths of the left and right borders and the widths of all boxes *left* the one under consideration. The right and bottom offsets are left undefined. The padding of the one box, which surrounds all the mentioned boxes, is taken into account by assuming, that the top margin of the top-most box and the left margin of the left-most box is at least as wide as the padding of the surrounding box.

2.3 BAiT: Testing and Test Data Selection

Behavior-Adaptation in Testing (BAiT, [5–7]) is a toolset, that implements the test generation and execution process displayed in Fig. 3. The process starts from a specification of the IUT. In our case, such a specification is given in the formal specification language μCRL [10], which is based on a process algebra with abstract datatypes. In a first step, behavior and data are separately extracted from the specification. The behavior part forms the abstract specification, which is further used for the generation of parameterizable test cases. Data relations in the regarded system form the test oracle, which is later used to actually parameterize the generated test cases.

The test generation branch of the process regards behavior after having abstracted data using a so-named chaotic data abstraction [6, 7], by which variable parameters are replaced by a distinct constant *chaos*. In doing so, we avoid the problem of state space explosion during the test generation process. The generation of parameterizable, abstract test cases is performed by the tool *Test*

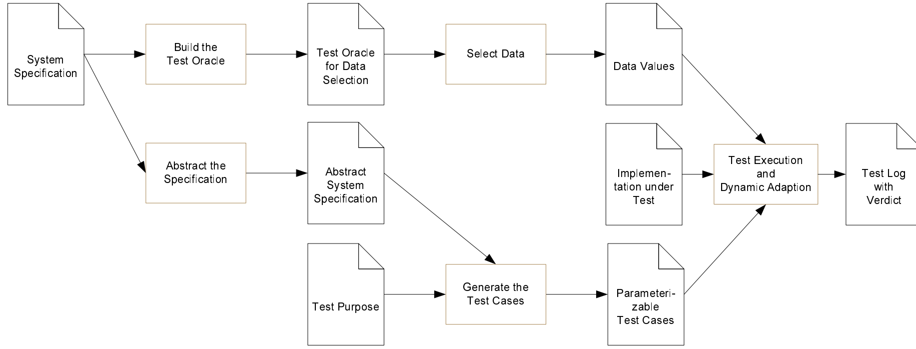

Fig. 3. Test Generation and Execution Process

Generation with Verification Techniques (TGV, [11]), which performs a guided search over the whole state space of the abstract specification. The test engineer can guide this process by providing test purposes, which sketch the focus of the test cases to be generated. The result of test case generation are parameterizable test cases, which can be instantiated with test data for test execution.

In parallel to test case generation, BAiT also prepares the later instantiation of the test cases with test data. Therefore, it generates a constraint logic program (CLP) from the system specification. This CLP holds all data interdependencies within the tested system and serves as a test oracle during test execution. Using the test oracle, data can be selected in order to instantiate parameters for the generated test cases.

Once both the parameterizable test cases and the test oracle have been generated, the test can be performed against an IUT. Depending, whether the IUT reacts as expected or not, the test ends with a verdict *Pass*, *Fail* or *Inconc*.

In our setting, TGV does not generate a single test case, i.e. a single trace through the system's behavior, but a so-named complete test graph (CTG). Such a CTG provides a set of traces through the system, whose final states can be marked as *Pass* or *Inconc* states. If during test execution a *Pass* state is reached in the CTG, then the test *passes*, i.e. no failure could be found in the system. If an *Inconc* state is reached, then the system behaved according to its specification, however, the test purpose was not met. This can happen, if the test purpose disallows a particular behavior which is allowed in the system's specification. In this case, it is not clear, whether a failure has been found in the test run, or not. Finally, a test can lead to the verdict *Fail*, if the IUT shows behavior, which is not allowed according to the specification.

For test execution, BAiT first selects a single abstract trace to a *Pass* verdict from the CTG. This trace is then instantiated and executed in a step-wise manner. If the IUT diverts from the precomputed trace by sending unexpected data to the tester, the trace under consideration is pruned. Then, BAiT checks, whether the IUT is still in an allowed system state according to the test oracle. If this is not the case, the test run ends with a verdict *Fail*. If it is the case,

then BAiT tries to find an alternative trace to a *Pass* verdict. If this attempt does not succeed, the test ends with verdict *Inconc*. If a trace to *Pass* could be successfully executed to its end, then the test run ends with verdict *Pass*.

3 Objective of the Case Study

The objective of our case study is to apply BAiT to testing the implementation of the *Gecko* rendering engine. BAiT has originally been designed as a tool for the test of data-oriented reactive systems in general. In this paper, we report on a feasibility study to validate the applicability of BAiT to HTML rendering engines. These systems (or system components, resp.) are not reactive systems in the original sense.

Reactive systems are based on events being sent forth and back between several systems. These events can be parameterized with data. A rendering engine works differently: It is document-centered, i.e. it receives a document and renders it. While sending the document to the rendering engine is still comparable to the reactive systems described above, rendering this document is not. It is no reaction in the original sense, since no evaluable events are sent back from the IUT. The only event sent back from the renderer states, that rendering is finished, but it does not contain the actual result of rendering. In many cases, the result of rendering must even be evaluated visually, while for some aspects, the relevant information can be retrieved from the rendering engine and can be computer-processed.

Such an aspect are the positions of `div`-boxes to which we will restrict the test. Rules for positioning such boxes are defined in the CSS Box Model, which has been described in Sect. 2.2. We will, however, not consider the full box model, but restrict to a fragment of it.

First of all, we concentrate on the settings `absolute` and `relative` with a binding to the top left corner of the web page for the possible positioning of boxes. Secondly, we consider empty boxes of an explicitly defined width and height only in order to keep the results of rendering predictable by the test oracle. Boxes filled with content may lead to overflowing content which results in a correction by the rendering engine based on information about the viewport dimensions and the used font dimensions. Treating those details in this feasibility study would not be purposeful and furthermore would have required arithmetic division operations, which would complicate the specification in μCRL. For this reason, we do not regard (overflowing) content in this case study.

Thirdly, we limit the possible scales used in the design definition of a `div`-box. Normally, the position and size of a `div`-box is determined by distances, which can be defined in a variety of scales. Some of those some are absolute (pixels, didot points, pico points, inches, millimeters and centimeters) and other are relative to either the actual font setting (scales em and ex) or the rest of the page layout (percentages or the `auto` setting). In this paper, we only consider absolute lengths of scale `px` (pixel) in order to avoid scale conversions.

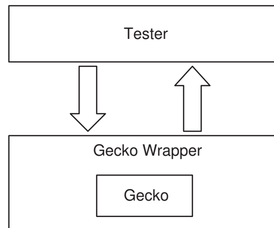

Fig. 4. Test Environment

Finally, we use a "flat" model in our case study rather than one, which resembles the whole nested structure of a web page. This means, that we regard only a distinct box *testbox* and its absolute position on the web page. When we add another box to the web page, then we recompute the position of *testbox* as it has changed due to the newly added other box. This means for instance, that, if we add another box above the regarded *testbox*, the top offset of *testbox* is recomputed as the summation of the previous top offset, the top and bottom margins of the new box, the top and bottom border width of the new box and the new box's height. By doing so, we can easily keep track of the position of the regarded *testbox* without having to keep the whole HTML document structure in our model.

Apart from the applicability of BAiT to HTML rendering engines in particular, we also aim at two other targets with this case study. On the one hand, we want to test the adaptability of BAiT to nondeterministic behavior of the IUT further by introducing some artificial nondeterminism w.r.t. to the system's feedback about the rendered boxes. On the other hand, we want to regard the feasibility of μCRL as a language for the design of test purposes.

4 Realizing the Test Environment

In order to test Gecko, we first had to create a test environment. This environment, as schematically depicted in Fig. 4, consists of a tester and an implementation under test (IUT). The tester is in our case the tool BAiT. The IUT is a component named Gecko Wrapper, which wraps Gecko internally. Both the tester and the IUT are Java components which communicate with each other using bidirectional procedure-based communication.

In order to generate and run the tests following the schema from Fig. 3, we also need a system specification of the CSS box model for Gecko and a test purpose to sketch out the later test cases. While the design of the tests and hence also that of the test purposes will be the topic of Sect. 5, we will in the remainder of this section discuss the specification of the boxmodel. Furthermore, we will give some details on the Gecko Wrapper.

Action	Functionality
Input actions:	
`resetBoxes`	Wipes all boxes and starts again with a fresh document.
`setupTestbox`	Defines the distinctly regarded test box.
`putBoxRelative`	Puts a box relatively to the other boxes yet defined in the actual HTML document. It can be defined, whether this box appears left of, right of, above, beneath or around all yet defined boxes. Finally, the measurements can be defined.
`putBoxAbsolute`	Puts a box with an absolute position.
`render`	Renders the actually defined document and starts returning results (offsets, see below)
Output actions:	
`offsetLeft`	Returns absolute left offset of test box.
`offsetRight`	Returns absolute right offset of test box.
`offsetTop`	Returns absolute top offset of test box.
`offsetBottom`	Returns absolute bottom offset of test box.

Table 1. Actions for the CSS box model

4.1 Modelling CSS in μCRL

We modeled a fragment of the CSS box model with the limitations from Sect. 3 in μCRL [10]. The modeled fragment of CSS allows to position boxes relative to each other or absolutely. In our model, recursive structures of boxes are flattened by regarding only one distinct box and its position, rather than a structure of boxes. Whenever a box is added, only the consequences on the position of the regarded box are computed and applied.

While rendering a web page is in principle a document-centered task, our specification of the box model is behavior-oriented. Hence, we defined a set of input actions, which allow us to put boxes into a box structure. Furthermore, we defined some output actions, which provide information about the current offset of the regarded box to the tester. The actions are defined in Table 1.

The system behavior for the CSS box model is specified as follows: As a first step, a testbox must be set up (`setupTestbox`). The action `setupTestbox` accepts parameters, which determine the box's width and height. Other boxes can be put in the vicinity of this testbox using the actions `putBoxRelative` and `putBoxAbsolute` in any order. The action `putBoxRelative` accepts 15 parameters: The first one determines, whether the box appears above, below, left, right or around the other boxes, which have yet been inserted into the web page. This parameter is named "position", but is actually not related the the `position`-property of CSS. The next two parameters determine the box's width and height. The last 12 parameters, finally, define the width of the box's padding, border and margin as depicted in Fig. 2. The action `putBoxAbsolute` only accepts seven parameters. The first three are identical with those from `putBoxRelative`, while the remaining four parameters define the box's absolute position on the page w.r.t. the four margins of the web page.

Any of these three actions can be followed by an action `resetBoxes` in order to delete all boxes and start from scratch, or by an action `render`. In this case, the IUT renders the defined structure of `div`-boxes. Afterwards, the different actual values for the offsets of the testbox (left, right, top, bottom offset) are returned by the IUT in an arbitrary order.

As we described in Sect. 3, we only regard a distinct box *testbox* in the model, whose position we recompute each time another box is added to the HTML test document. The actions `putBoxAbsolute` and `putBoxRelative` change this position in the described way. Below, we give the μCRL code, which relates to the behavior of `putBoxRelative`:

```
...
PrepareRendering(position:PositionType,relation:RelationType,
   width:Nat,height:Nat,offsetLeft:Nat,offsetRight:Nat,
   offsetTop:Nat,offsetBottom:Nat) =
...
 + sum(pposition:PositionType,sum(pwidth:Nat,sum(pheight:Nat,...,
   putBoxRelative(pposition,pwidth,pheight,pmarginleft,
       pmarginright,pmargintop,pmarginbottom,pborderleft,...).
   PositionBoxRelative(position,relation,width,height,...)...)))
...
PositionBoxRelative(position:PositionType,relation:RelationType,
   width:Nat,height:Nat,...) =
   tau.PrepareRendering(...,offsetLeft+pmarginLeft+
       pborderLeft+ppaddingLeft+pwidth+ppaddingRight+
       pborderRight+pmarginRight,...)
    <|and(eq(relation,relative),eq(pposition,left))|>delta
 + tau.PrepareRendering(...,offsetLeft,0,offsetTop,offsetBottom)
     <|and(eq(relation,relative),eq(pposition,right))|>delta
 + tau.PrepareRendering(...,offsetTop+pmarginTop+pborderTop+
        ppaddingTop+pheight+ppaddingBottom+pborderBottom+
        pmarginBottom,offsetBottom)
     <|and(eq(relation,relative),eq(pposition,top))|>delta
 + tau.PrepareRendering(...,offsetLeft,offsetRight,offsetTop,0)
     <|and(eq(relation,relative),eq(pposition,bottom))|>delta
...
```

The above fragment of our specification shows the definition of the action `putBoxRelative` within a process `PrepareRendering`. When `putBoxRelative` has been invoked, the system enters another process, `PositionBoxRelative`. After a τ-step, the system leaves `PositionBoxRelative` and goes back to `PrepareRendering`. While doing so, the new position of the test box is computed depending on the value of the variable `pposition` of the newly positioned box. This leads to a case distinction depending on the position of the new box.

4.2 Wrapping Mozilla Gecko

Mozilla Gecko can be embedded into custom applications as a component, which can be programmed using its XPCOM interfaces. The Cross Platform Component Object Model (XPCOM, [13]) is an unmanaged component framework, which is used for the Mozilla software products. Gecko can be embedded into Java applications. In order to do so, one can either instantiate it directly via its XPCOM interfaces or embed it indirectly via the `Browser` component of the Standard Widget Toolkit (SWT)[8].

We implemented a wrapper for Gecko in Java using SWT. The wrapper receives all actions which place boxes and builds up an internal structure for a test web page. On action `render`, the wrapper generates actual HTML and CSS code and sends this code to the renderer. A window is opened for rendering.

When rendering is finished, the renderer is queried for the offsets. For this procedure, we followed an existing code example[9]. In order to query an offset, a short piece of JavaScript code is generated and executed within the web browsing component. This piece of code internally queries for the respective offsets and writes the result to an (invisible) status bar. When this has happened, the wrapper can read the value from this status bar in order to store it, and the next piece of JavaScript is generated and executed (one execution per one of the four offset parameters). After all offsets have been queried, the tester is informed by the actions `offsetLeft`, `offsetRight`, `offsetTop` and `offsetBottom` in a random order. We chose for a random order in order to test the adaptation of BAiT to nondeterministic behavior of the IUT, as we have described in Sect. 3.

5 Running the Tests

5.1 Design of the Test Cases

In the BAiT approach, test generation is based on the tool TGV [11]. This tool takes as input the system specification in the form of an LTS as well as a test purpose. In a first step, prior to test case generation, we applied data abstraction on the specification of the system, in order to avoid space explosion induced by the many unrestricted numerical parameters of the input actions `setupTestbox`, `putBoxAbsolute` and `putBoxRelative`.

The second step was to design test purposes. We designed two test purposes, of which one traditionally directly as an LTS, while the second one was specified in μCRL as was the system itself. According to the first test purpose, we set up a testbox, put at least one more (relatively positioned) box in its vicinity and render the resulting HTML document. Having done so, we expect the system to return at least the top offset of the testbox. The second test purpose is designed a bit differently, since we still wanted to experiment more with BAiT's capability

[8] `http://www.eclipse.org/swt`

[9] `http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.eclipse.swt.snippets/src/org/eclipse/swt/snippets/Snippet160.java`

Test purpose in μCRL:

```
% datatype definitions and action definitions skipped
proc
  PASS = ACCEPT.PASS
  FAIL = REFUSE.FAIL

  PutBox = render + putBoxRelative.PutBox
  TP = setupTestbox.putBoxRelative.PutBox.
         (offsetLeft.PASS + offsetBottom.FAIL)

init TP
```

Resulting test purpose as LTS before adding placeholders for action parameters:

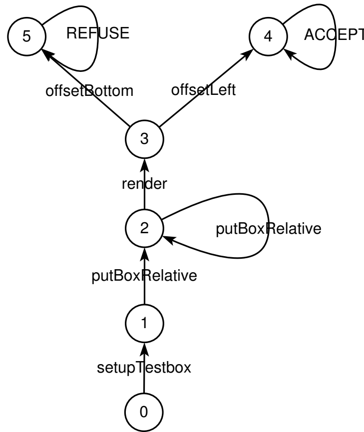

Fig. 5. A test purpose both in μCRL and as an LTS

of behavior-adaptation during a test run. For this purpose, the test purpose was designed to expect at least the left offset of the testbox and to refuse an action `offsetBottom` following directly on the `render` action. This refusal in combination with the absolutely random order of `offset`-events from the IUT leads to more situations in which BAiT will be led into a trace to an *Inconc* verdict, from which it will try to find an alternative trace to a *Pass* verdict. BAiT will, however, never find such a trace and it will have to give up terminating with verdict *Inconc*. Since the generated test cases can contain loops, BAiT might search for a trace to *Pass* without ever terminating. This issue has been solved by introducing a configuration option for BAiT, which defines the maximum amount of traces to search for before giving up and assigning *Inconc*. This second test purpose is shown in Fig. 5 as a μCRL specification and an LTS.

In a third step, we generated complete test graphs (CTGs) with TGV. The abstracted system specification as input to TGV was quite manageable with its

17 states and 57 transitions, so that the generation process took place within negligible time. For the first test purpose, generation resulted in a CTG with 25 states and 70 transitions. The second test purpose put more restrictions on the behavior of the IUT during the test, so the number of transitions in the resulting CTG was reduced to 59; the number of states increased slightly to 28. In general, these numbers are relatively low, a circumstance which does not astonish if one keeps in mind, that we regard only the behavior of a highly data-intensive system after data abstraction. The main work, as we had already remarked earlier, is the data selection during test execution.

5.2 Test Execution

Based on the generated CTGs, we ran some tests with BAiT and the Gecko wrapper. We used the default trace search algorithm of BAiT in order to select test traces through the CTGs. This algorithm searches only for traces to *Pass*, using a breadth-first search. Having automatically selected a trace to *Pass*, we then selected data for the different parameters of the box positioning actions and executed the trace. During the different test runs, we found a few failures. However, those failures were induced by faults in the used model rather than by the IUT itself. After having eliminated the faults, we did not find any more failures in the IUT.

As expected, the test runs based on the first CTG always ended in a *Pass* verdict, after we had corrected the model. The test runs based on the second CTG, randomly went to a *Pass* or an *Inconc* verdict. This behavior was dependent on whether the wrapper returned an `offsetBottom` event before (*Inconc*) or after (*Pass*) the `offsetLeft` event (cf. the description of the second test purpose). Since the order of events was implemented in the wrapper to be random, the assignment of verdicts was also as expected a priori.

6 Conclusion

In this paper, we presented the application of an existing approach for conformance testing, Behavior-Adaptation in Testing (BAiT), to the test of the HTML rendering engine Mozilla Gecko. This is a new application for BAiT, since the toolset was originally designed to treat nondeterministic reactive, and thus action-based, systems with data. We modeled a fragment of the CSS box model in the formal specification language μCRL, implemented a wrapper for Mozilla Gecko in order to be able to apply the BAiT for test execution, and designed some test cases following the BAiT approach. In a controlled experiment, we validated the applicability of BAiT to document-centered HTML rendering.

We designed the case study as a feasibility study in order to do a first evaluation for further experiments with BAiT and Gecko. Focusing on a subset of aspects and leaving out things like handling of boxes with overflowing content, decreased the design effort of the model of our fragment of the CSS box model and the implementation effort w.r.t. the wrapper for Gecko significantly.

Our experiences with the design of test purposes were twofold. On the one hand the behavior-oriented part of the design was very easy. On the other hand, this means that most of the test design is induced by data and thus actually happens during the test run itself and can be computed automatically. We did not encounter any problems at this point. The main issue was the fact, that the data parameters for the configuration of the `div`-boxes were mostly independent of each other. This did not constitute a problem by itself, however, the capabilities of BAiT guiding test data selection by precomputing the ranges of data parameters could not be beneficial in this case. Running the second test case, we encountered, that BAiT's behavior adaptation can easily lead to test runs, which do not terminate. In this case, a careful configuration of the search threshold of BAiT was necessary to prevent infinite test runs while at the same time avoiding superfluous *Inconc* verdicts. In real life testing of Gecko, this would not be an issue, since the superfluous *Inconc* verdicts are induced by the nondeterminism, which we artificially built into the IUT.

The feasibility study described in this paper was successful and forms an important step towards a fully automated model-based test approach for HTML rendering engines using BAiT. Compared to static test suites, which serve the document-centered HTML rendering process, our behavior-oriented approach has the advantage of a higher flexibility w.r.t. data parameters and the tested document structure. With a wrapper for Gecko and a behavior-oriented approach to test the IUT, it is far easier to generate a variety of different HTML test documents, which cover different aspects of the IUT. Special expertise is only necessary in order to provide BAiT with a formal specification of the CSS box model. Test data of a sufficient quality can – also for critical values causing overflowing boxes – can, for instance, easily be selected by the developers of the rendering engine themselves.

From the reached state, we can now extend the work in some respect. In order to do a full model-based test of Gecko (or other rendering engines) w.r.t. the CSS box model, it is necessary to extend the formal specification to the whole model. Since the model in [15] is given in natural language rather than a formal notation, it might quite likely be incomplete, ambiguous and may contain semantic variation points. Such aspects would complicate an attempt to formalize the model. Another, technical, issue concerning the conformance test with the full box model is the treatment of floating point datatypes rather than integers. Not only, that the specification language used in this case study does not directly support floating point datatypes, also some aspects of test data selection must be considered. At the moment, we consider output from the IUT being exactly equal to the expected output in order to pass the test. In a floating point setting, we would have to consider the output to be not necessarily exactly equal to the expected output, but to be within a particular margin around our expectations. A last issue to consider w.r.t. a full formal model of the CSS box model is, that this extended model may not be flattened anymore like the one we used in this case study. It rather has to reflect the nested structure of boxes

on the rendered web page in its system status. This issue also affects the test data selection approach.

References

1. B. K. Aichernig, B. Peischl, M. Weiglhofer, and F. Wotawa. Test Purpose Generation in an Industrial Application. In *Proc. of the 3rd Intl. Workshop on Advances in Model-based Testing*, pages 115–125. ACM, 2007.
2. A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation: A Simple Experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *12th Intl. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.
3. M. Born, I. Schieferdecker, H.-G. Gross, and P. Santos. Model-driven Development and Testing – A Case Study. In M. J. van Sinderen and L. F. Pires, editors, *1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, number TR-CTIT-04-12 in CTIT Technical Report, pages 97–104, Enschede, 2004.
4. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.
5. J. R. Calamé. Adaptive Test Case Execution in Practice. Technical Report SEN-R0703, Centrum voor Wiskunde en Informatica, June 2007.
6. J. R. Calamé, N. Ioustinova, and J. C. van de Pol. Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proc. of the Doctoral Symposium affiliated with the 5th Intl. Conf. on Integrated Formal Methods (IFM 2005)*, volume 191 of *Electronic Notes in Computer Science*, pages 25–48. Elsevier, October 2007.
7. J. R. Calamé, N. Ioustinova, J. C. van de Pol, and N. Sidorova. Data Abstraction and Constraint Solving for Conformance Testing. In *Proc. of the 12th Asia-Pacific Software Engineering Conf. (APSEC'05)*, pages 541–548. IEEE Press, 2005.
8. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated Test and Oracle Generation for Smart-Card Applications. In *Intl. Conf. on Research in Smart Cards (e-Smart'01)*, volume 2140 of *LNCS*, pages 58–70, 2001.
9. W. Geurts, K. Wijbrans, and J. Tretmans. Testing and Formal Methods – Bos Project Case Study. In *EuroSTAR'98: 6th European Intl. Conf. on Software Testing, Analysis & Review*, pages 215–229, 1998.
10. J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse, C. Verhoef, and S. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62, Berlin, 1994. Springer.
11. C. Jard and T. Jéron. TGV: Theory, Principles and Algorithms. *Intl. Journ. on Software Tools for Technology Transfer*, 7(4):297–315, 2005.
12. J. Tretmans. Test Generation with Inputs, Outputs, and Repetitive Quiescence. *Software - Concepts & Tools*, 17(3):103–120, 1996.
13. D. Turner and I. Oeschger. *Creating XPCOM Components*, 2003.
14. W3C. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). online: `http://www.w3.org/TR/2002/REC-xhtml1-20020801`, August 2002. W3C Recommendation.
15. W3C. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. online: `http://www.w3.org/TR/2007/CR-CSS21-20070719`, July 2007. W3C Candidate Recommendation.
16. N. C. Zakas, J. McPeak, and J. Fawcett. *Professional AJAX*. Wrox Press, 2006.