1 Combinatorial Testing Applied

In Chapter 1, we introduced the Interaction Rule, and noted that if all failures can be triggered by *t*-way combinations or less, then testing all *t*-way combinations is in some sense equivalent to exhaustive testing. We have referred to this approach as *pseudo-exhaustive* testing [91]: clearly it is not exhaustive because we do not cover all possible inputs and issues such as timing and concurrency may be missed. But the ability to test all combinations of (discretized) values, up to the interaction strength actually involved in failures, provides a powerful assurance method. Can this approach really detect as many faults as completely exhaustive testing? Although it will take years of experience with combinatorial testing to give a full answer, this chapter summarizes two studies of practical problems where combinatorial and exhaustive approaches were compared. Results indicate that combinatorial testing can indeed find faults as well as exhaustive testing, using only a fraction of the tests.

1.1 The Document Object Model

(Carmelo Montanez, Rick Kuhn, Mary Brady, Rick Rivello, Jenise Reyes, Michael Powers)

The Document Object Model (DOM) [197] is a standardized method for representing and interacting with components of XML, HTML, and XHTML documents. DOM lets programs and scripts access and update the content, structure, and style of documents dynamically, making it easier to produce web applications in which pages are accessed non-sequentially. DOM is standardized by the World Wide Web Consortium (W3C). Since its origination in 1996 as a convention for accessing and modifying parts of Javascript web pages (known now as DOM Level 0), DOM has evolved as a series of standards offering progressively greater capabilities. Level 1 introduced a model that allowed changing any part of the HTML document, and Level 2 added support for XML namespaces, load and save, cascading style sheets (CSS), traversing the document, and working with ranges of content. Level 3 brings additional features, including keyboard event handling.

DOM Level 3 Events [197] is a W3C Working Draft being written by the Web Applications Working group. Implemented in browsers, it is a generic platform and language neutral event system that allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event. This work builds on the previous Document Object Model Level 2 events specifications. There are two basic goals in the design of DOM Level 3 Events. The first goal is to design an event system that allows registration of event listeners and describes an event flow through a tree structure. The second goal is to provide a common subset of the current event system used on DOM Level 3 Events browsers.

Constructing Tests for DOM Events

Full exhaustive testing (of discretized inputs) was initially used for DOM conformance, more than 36,000 tests.

DOM browser implementations typically contain tens of thousands of source lines of code. To help ensure successful implementations of this complex standard, NIST developed the DOM Conformance Test Suites [198], which include tests for many DOM components. Early DOM tests were hand-coded in a test language, then processed to produce ECMAScript and Java. In the current version of the test suites, tests are specified in an XML grammar, allowing easy mapping from specification to a variety of language bindings. Because the grammar is generated automatically [125] from the DOM specs, tests can be constructed quickly and correctly. Output of the test generation process includes the following components, which implementers can use in testing their product for DOM interoperability:

- Tests in the XML representation language,
- XSLT stylesheets necessary to generate the Java and ECMA Script bindings,
- Generated executable code.

Tests for 35 (out of 36) DOM Events were generated. The specification defines each event with an Interface Definition Language (IDL), which in turn defines a number of functions for each event. A typical function can have anywhere from one to fifteen parameters. The API for each function is defined as an XML infoset, which specifies the abstract data model of the XML document using a predefined set of tags. The XML infosets were programmatically generated through a Java application. Since the IDL definition could be accessed directly from the specs web site; the web address was given as input to the Java application. This way the application could read and traverse them extracting just the information of interest. In this case the function names and their respective parameters, argument names, etc., which became part of the XML file that was used to feed a test generation tool to automatically create the DOM Level 3 tests.

Conventional category partitioning was used to select representative values for non-Boolean parameters. The initial test set was exhaustive across the equivalence classes, producing 36,626 tests that exercised all possible combinations of equivalence class elements. Note that this is not fully exhaustive – all possible value combinations – because such a test suite is generally intractable with continuous-valued parameters. It is however exhaustive with respect to the equivalence class elements. Thus this test suite will be referred to as the *exhaustive test suite* in the remainder of this report. Two different implementations were tested. The DOM events and number of tests for each are shown in Table 1. This set of exhaustive tests detected a total of 72 failures. Automated tools made it possible to construct tests programatically, greatly reducing the effort required for testing. However, human intervention is required to run individual tests and evaluate test results, so the conformance testing team sought ways to reduce the number of tests required without sacrificing quality.

Combinatorial Testing Approach

Because the DOM test suite had already been applied with exhaustive (across equivalence values) tests against a variety of implementations, it provided a valuable opportunity to evaluate combinatorial testing on real-world software. If results showed that a much smaller test suite could achieve the same level of fault detection, then testing could be done at much less cost in staff time and resources. An obvious critical question in using this approach is – *what level of t-way interaction is necessary*? Can all faults be detected with 2-way (pairwise) tests, or does the application require 3-way, 4-way or higher strength tests? This work helped to address these questions as well.

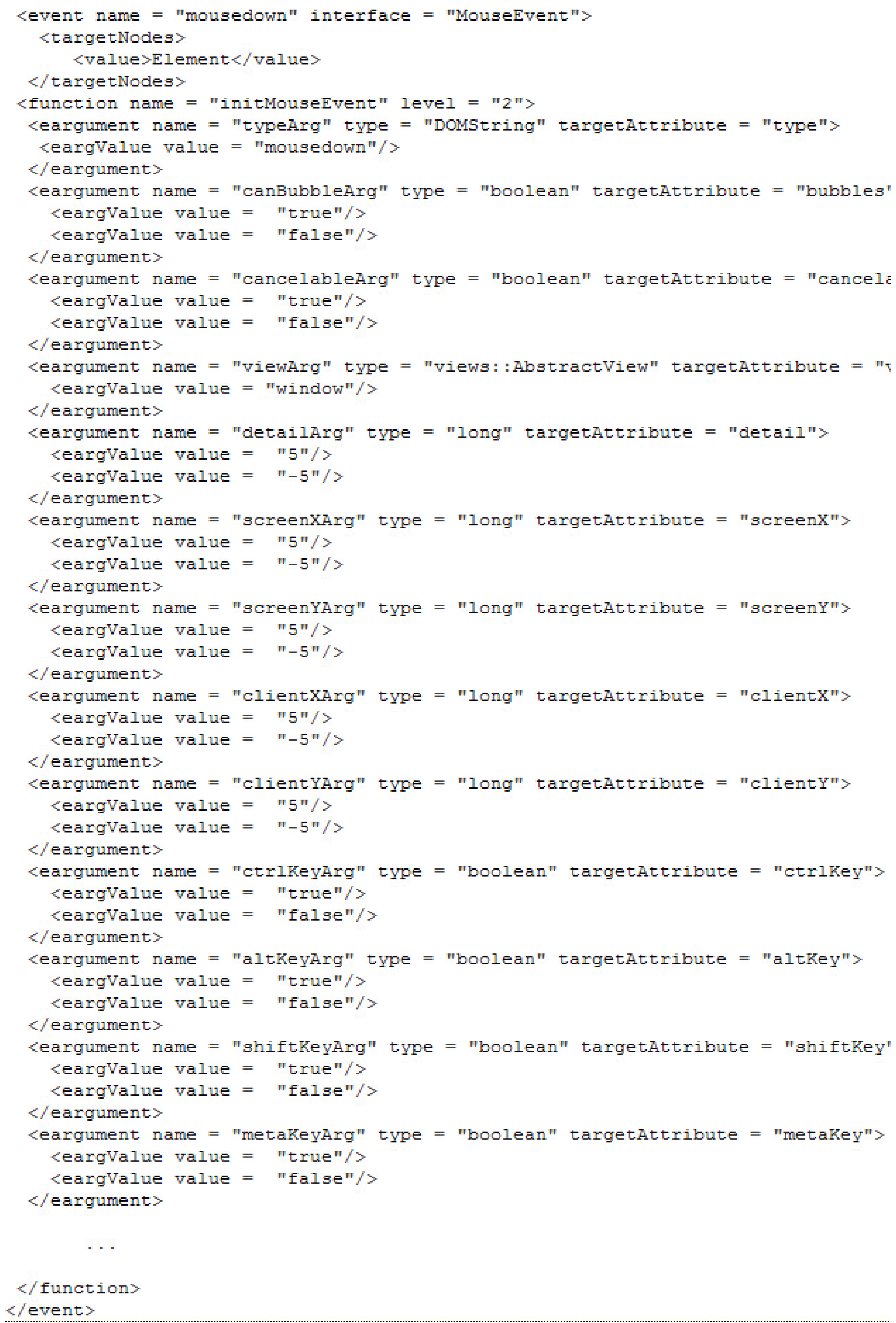
To investigate the effectiveness of combinatorial testing, covering arrays of 2-way through 6-way tests were produced, using ACTS. Using *t*-way combinations can significantly reduce the number of tests as compared with exhaustive. For example, the `mousedown` event (Figure 1) requires 4,352 tests if all combinations are to be realized. Combinatorial testing reduces the set to 86 tests for 4-way coverage.

Table 2 details the number of parameters and number of tests produced for each of the 35 DOM events, for $t = 2$ through 6. That is, the tests covered all 2-way through 6-way combinations of values. Note that for events with few parameters, the number of tests is the same for the original test suite (Table 1) and combinatorial for various levels of *t*. For example, 12 tests were produced for Abort in the original and also for combinatorial testing at $t = 3$ to 6. This is because producing all *n*-way combinations for *n* variables is simply all possible combinations of these *n* variables, and Abort has 3 variables. This situation is not unusual when testing configurations with a limited number of values for each parameter.

For nine of the 35 events (two Click events, six Mouse events, and Wheel), all combinations are not covered even with 6-way tests. For these events, combinatorial testing provides a significant gain in efficiency.

Event Name	Param.	Tests
Abort	3	12
Blur	5	24
Click	15	4352
Change	3	12
dblClick	15	4352
DOMActivate	5	24
DOMAttrModified	8	16
DOMCharacterDataModified	8	64
DOMElementNameChanged	6	8
DOMFocusIn	5	24
DOMFocusOut	5	24
DOMNodeInserted	8	128
DOMNodeInsertedIntoDocument	8	128
DOMNodeRemoved	8	128
DOMNodeRemovedFromDocument	8	128
DOMSubTreeModified	8	64
Error	3	12
Focus	5	24
KeyDown	1	17
KeyUp	1	17
Load	3	24
MouseDown	15	4352
MouseMove	15	4352
MouseOut	15	4352
MouseOver	15	4352
MouseUp	15	4352
MouseWheel	14	1024
Reset	3	12
Resize	5	48
Scroll	5	48
Select	3	12
Submit	3	12
TextInput	5	8
Unload	3	24
Wheel	15	4096
Total Tests		36626

Table 1. DOM Level 3 Events Tests – Exhaustive



```xml
 <event name = "mousedown" interface = "MouseEvent">
   <targetNodes>
      <value>Element</value>
  </targetNodes>
 <function name = "initMouseEvent" level = "2">
  <eargument name = "typeArg" type = "DOMString" targetAttribute = "type">
   <eargValue value = "mousedown"/>
  </eargument>
  <eargument name = "canBubbleArg" type = "boolean" targetAttribute = "bubbles'
    <eargValue value =  "true"/>
    <eargValue value =  "false"/>
  </eargument>
  <eargument name = "cancelableArg" type = "boolean" targetAttribute = "cancel
    <eargValue value =  "true"/>
    <eargValue value =  "false"/>
  </eargument>
  <eargument name = "viewArg" type = "views::AbstractView" targetAttribute = "
    <eargValue value = "window"/>
  </eargument>
  <eargument name = "detailArg" type = "long" targetAttribute = "detail">
    <eargValue value =  "5"/>
    <eargValue value =  "-5"/>
  </eargument>
  <eargument name = "screenXArg" type = "long" targetAttribute = "screenX">
    <eargValue value =  "5"/>
    <eargValue value =  "-5"/>
  </eargument>
  <eargument name = "screenYArg" type = "long" targetAttribute = "screenY">
    <eargValue value =  "5"/>
    <eargValue value =  "-5"/>
  </eargument>
  <eargument name = "clientXArg" type = "long" targetAttribute = "clientX">
    <eargValue value =  "5"/>
    <eargValue value =  "-5"/>
  </eargument>
  <eargument name = "clientYArg" type = "long" targetAttribute = "clientY">
    <eargValue value =  "5"/>
    <eargValue value =  "-5"/>
  </eargument>
  <eargument name = "ctrlKeyArg" type = "boolean" targetAttribute = "ctrlKey">
    <eargValue value =  "true"/>
    <eargValue value =  "false"/>
  </eargument>
  <eargument name = "altKeyArg" type = "boolean" targetAttribute = "altKey">
    <eargValue value =  "true"/>
    <eargValue value =  "false"/>
  </eargument>
  <eargument name = "shiftKeyArg" type = "boolean" targetAttribute = "shiftKey'
    <eargValue value =  "true"/>
    <eargValue value =  "false"/>
  </eargument>
  <eargument name = "metaKeyArg" type = "boolean" targetAttribute = "metaKey">
    <eargValue value =  "true"/>
    <eargValue value =  "false"/>
  </eargument>

       ...

 </function>
</event>
```

Figure 1.   XML infosets generated from IDLs used as input to Test Accelerator.

Event Name	Num param	2-way Tests	3-way Tests	4-way Tests	5-way Tests	6-way Tests
Abort	3	8	12	12	12	12
Blur	5	10	16	24	24	24
Click	15	18	40	86	188	353
Change	3	8	12	12	12	12
dblClick	15	18	40	86	188	353
DOMActivate	5	10	16	24	24	24
DOMAttrModified	8	8	16	16	16	16
DOMCharacterDataModified	8	32	62	64	64	64
DOMElementNameChanged	6	8	8	8	8	8
DOMFocusIn	5	10	16	24	24	24
DOMFocusOut	5	10	16	24	24	24
DOMNodeInserted	8	64	128	128	128	128
DOMNodeInsertedIntoDocument	8	64	128	128	128	128
DOMNodeRemoved	8	64	128	128	128	128
DOMNodeRemovedFromDocument	8	64	128	128	128	128
DOMSubTreeModified	8	32	64	64	64	64
Error	3	8	12	12	12	12
Focus	5	10	16	24	24	24
KeyDown	1	9	17	17	17	17
KeyUp	1	9	17	17	17	17
Load	3	16	24	24	24	24
MouseDown	15	18	40	86	188	353
MouseMove	15	18	40	86	188	353
MouseOut	15	18	40	86	188	353
MouseOver	15	18	40	86	188	353
MouseUp	15	18	40	86	188	353
MouseWheel	14	16	40	82	170	308
Reset	3	8	12	12	12	12
Resize	5	20	32	48	48	48
Scroll	5	20	32	48	48	48
Select	3	8	12	12	12	12
Submit	3	8	12	12	12	12
TextInput	5	8	8	8	8	8
Unload	3	16	12	24	24	24
Wheel	15	20	44	92	214	406
Total Tests		702	1342	1818	2742	4227

Table 2. DOM 3 Level Tests – Combinatorial

Test Results

4-way combinatorial testing found all DOM faults using 95% fewer tests than the original test suite.

Table 2 shows the faults detected for each event. All conditions flagged by the exhaustive test suite were also detected by three of the combinatorial testing scenarios (4, 5 and 6 way testing), which indicates that the implementation faults were triggered by 4-way interactions or less. Pairwise testing would have been inadequate for the DOM implementations, because 2-way and 3-way tests detected only 37.5% of the faults. As can be seen in Table 2, the exhaustive (all possible combinations) tests of equivalence class elements and the 4-way to 6-way combinatorial tests were equally successful in fault detection, indicating that exhaustive testing added no benefit. These findings are consistent with earlier studies [93] that showed that software faults are triggered by interactions of a small number of variables, for applications in a variety of domains. DOM testing was somewhat unusual in that exhaustive testing of equivalence class elements was possible at all. For most software, too many possible input combinations exist to cover even a tiny fraction of the exhaustive set, so combinatorial methods may be of even greater benefit for these.

t	**Tests**	**Pct of Original**	**Test Results**		
			Pass	**Fail**	**Not Run**
2	702	1.92%	202	27	473
3	1342	3.67%	786	27	529
4	1818	4.96%	437	72	1309
5	2742	7.49%	908	72	1762
6	4227	11.54%	1803	72	2352

Table 3. Comparison of t-way with exhaustive test set size.

The original test suite contained a total of 36,626 tests (Table 1) for all combinations of events, but after applying combinatorial testing, the set of tests is dramatically reduced depending on the level of *t*-way interactions tested, as shown in Table 3. Table 3 also shows results for 2-way through 6-way testing. Notice that although the number of tests that successfully execute varies among *t*-way combination, the number of failures remains a constant at t = 2 and 3, and at t = 4 to 6. The last column shows the tests that did not execute to completion, in almost all cases due to non-support of the feature under test.

DOM results were consistent with previous findings that the number of parameters interacting in failures is small (in this case 4-way). Comparing results of the DOM testing with previous data on *t*-way interaction failures (Figure 2), we can see that some DOM failures were more difficult to detect, in the sense that a smaller percentage of the total were found by 2-way and 3-way tests than for the other application domains. The unusual shape of the curve for DOM tests may result from the large number of parameters for which exhaustive coverage was reached (so that the number of tests remained constant after a certain point). There are thus two sets of events: a large set of 26 parameters with few possible values that could be covered exhaustively with 2-way or 3-way tests, and a smaller set with a larger input space (from 1024 to 4352). In particular, nine events (`click`, `dblClick`, `mouse` events, and `wheel`) all have the same input space size, with number of tests increasing at the same rate for each, while for the rest, exhaustive coverage is reached at either t=2 or t=3. The ability to compare results of previously-conducted exhaustive testing with combinatorial testing provides an added measure of confidence in the applicability of these methods to this type of interoperability testing.

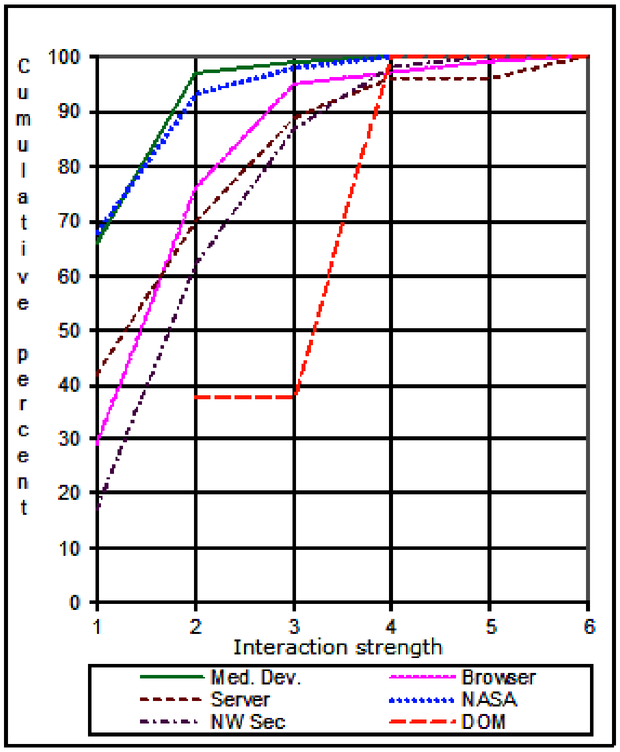


Figure 2. DOM compared with other applications

Cost and Practical Considerations

The DOM Events testing shows that combinatorial testing can significantly reduce the cost and time required for conformance testing for standards with characteristics similar to DOM. What is the appropriate interaction strength to use in this type of testing? Intuitively, it seems that if no additional faults are detected by *t*-way tests, then it may be reasonable to conduct additional testing only for *t*+1 interactions, but no greater if no additional faults are found at *t*+1. In empirical studies of software failures, the number of faults detected at $t > 2$ decreased monotonically with *t*, and the DOM testing results are consistent with this earlier finding. Following this strategy for the DOM testing would result in running 2-way tests through 5-way, then stopping because no additional faults were detected beyond the 4-way testing. Alternatively, given the apparent insufficient fault detection of pairwise testing, testers may prefer to standardize on a 4-way or higher level of interaction coverage. This option may be particularly attractive for an organization that produces a series of similar products and has enough experience to identify the most cost-effective level of testing. Even the relatively strong 4-way testing in this example was only 5% of the original test set size. Results in this study have been sufficiently promising for combinatorial methods to be applied in testing other interoperability standards.

1.2 Rich Web Applications

(Chad Maughan)