Test Automation to the Limit

Arie van Deursen
Delft University of Technology

Test Automation Day, 23 June, 2011
Outline

1. Background

2. Testing Web Apps to the Limit

3. Testing Plug-ins to the Limit

4. Discussion

Joint work with Ali Mesbah (UBC), Danny Roest (TU Delft), Michaela Greiler (TU Delft)
The TU Delft Software Engineering Research Group

Group size:
• ~30 fte, including
• ~20 full time researchers (PhD students, postdocs).

Teaching:
• Programming, projects
• ~20 MSc projects p/y in software engineering

Research:
• Software evolution
• Reverse engineering
• Software testing
• Software architecture
• End-user programming
• Software services
• Model-driven engineering
Test Automation?

- What benefits can your technique achieve?
- Within what limits is your technique successful?

- What’s going wrong if you move beyond those limits?
- *What can we do to move the limit?*
Part I: Testing Web Apps to the Limit

Asynchronous Javascript And XML
An Ajax Fault Model

- Stateful client;
- Asynchronous communication;
- Delta updates;
- Untyped JavaScript;
- Client-side DOM manipulation;
- Event handling;
- Timing;
- Back/forward, undo/redo
- Browser dependence (IE, FF, Chrome, ...)

- Focus on client side GUI logic.
- Different fault models / test strategies for server side / database / business etc. logic
Testing Web Applications

Traditional:
- Response – Request
- *No client side logic*

Capture & Playback:
- Selenium
- *Manual effort*

Can we push these limits?
**State:** DOM Tree

**Edge:** Event that is fired and causes a state transition
Steps Involved

• Finding (new) clickables
• Triggering events
• Comparing DOM trees
• Recurse
• Backtrack
• Input data generation
Implementation: Crawljax

• Open source: http://crawljax.com

• Java, maven, JUnit, Apache libraries
• Relies on webdriver to access the browser
  – IE, Firefox, Chrome

• Plugin-based architecture

• Contributions from NL, Canada, Japan, ...
• Used at Microsoft, Google, Fujitsu, eBay, ...
## Scalability Indicators

<table>
<thead>
<tr>
<th>Case</th>
<th>DOM string size (byte)</th>
<th>Candidate Clickables</th>
<th>Detected Clickables</th>
<th>Detected States</th>
<th>Crawl Time (s)</th>
<th>Depth</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>4590</td>
<td>540</td>
<td>16</td>
<td>16</td>
<td>14</td>
<td>3</td>
<td>A, DIV, SPAN, IMG</td>
</tr>
<tr>
<td>C2</td>
<td>24636</td>
<td>1813</td>
<td>33</td>
<td>34</td>
<td>26</td>
<td>2</td>
<td>A, IMG</td>
</tr>
<tr>
<td>C3</td>
<td>262505</td>
<td>150</td>
<td>148</td>
<td>148</td>
<td>498</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>19247</td>
<td>1101</td>
<td>1071</td>
<td></td>
<td>5012</td>
<td>2</td>
<td>A, TD</td>
</tr>
<tr>
<td>C4</td>
<td>40282</td>
<td>3808</td>
<td>55</td>
<td>56</td>
<td>77</td>
<td>2</td>
<td>A, DIV, INPUT, IMG</td>
</tr>
<tr>
<td>C5</td>
<td>165411</td>
<td>267</td>
<td>267</td>
<td>145</td>
<td>806</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>32365</td>
<td>1554</td>
<td>1234</td>
<td></td>
<td>6436</td>
<td>2</td>
<td>A, DIV</td>
</tr>
<tr>
<td>C6</td>
<td>134404</td>
<td>6972</td>
<td>83</td>
<td>79</td>
<td>701</td>
<td>1</td>
<td>A, DIV</td>
</tr>
</tbody>
</table>
Crawling for *Testing* Purposes?

- Crawling = Testing?
- Oracle?
  - Browser eats all issues
- Leverage derived model
Invariants as Oracles

• Generic, on the DOM-tree state
  – validity, error messages in DOM, accessibility, link discoverability, security...

• Generic, between successive DOM states
  – Consistent back-button

• Application-specific (design) invariants:
  – Constraints on specific element and attribute relations in particular DOM states
## Example Invariants

### Pointing to Faults

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Inv. Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>THEORGANIZER</td>
<td>The Year View state //img[contains(@src, 'head_yearView')] contains at least one appointment item //input[@id='edit']</td>
<td>XPath expr. state invariant</td>
</tr>
<tr>
<td>THEORGANIZER</td>
<td>'Failed to populate the list properly!'</td>
<td>Generic ('Fail%') DOM invariant</td>
</tr>
<tr>
<td>THEORGANIZER</td>
<td>Appointment item (view) structure</td>
<td>Reg. expr. state invariant</td>
</tr>
<tr>
<td>THEORGANIZER</td>
<td>Clicking on the logoff button //img[contains(@id, 'logoff')] results in a state with //div[contains(text(), 'You have logged out')]</td>
<td>application-specific SM inv.</td>
</tr>
<tr>
<td>THEORGANIZER</td>
<td>Clicking on //img[@id=&quot;X&quot;] results in a state with //img[contains(@src, 'head_X')], where X is a string</td>
<td>application-specific SM inv.</td>
</tr>
<tr>
<td>TASKFREAK</td>
<td>Top level body contains div[@id='header'], div[@id='container'], and at most one div[@id='calendar']</td>
<td>XPath expr., state invariant</td>
</tr>
<tr>
<td>TASKFREAK</td>
<td>All pages displaying current tasks via table[@id='taskSheet'] match a given template</td>
<td>Reg. expr. state invariant</td>
</tr>
<tr>
<td>TASKFREAK</td>
<td>Reload button in any state found via img[@id='frk-status'] should lead to state displaying current tasks</td>
<td>application-specific SM inv.</td>
</tr>
<tr>
<td>HILIST</td>
<td>Contact template, shown in Figure 13</td>
<td>Reg. expr. (template) state invariant</td>
</tr>
<tr>
<td>THETUNNEL</td>
<td>global variable alive is true during the game, and false after player fails</td>
<td>JAVA SCRIPT invariant</td>
</tr>
<tr>
<td>THETUNNEL</td>
<td>position of ship must be 32 times higher than the wall ship_x + 32 &gt;= right_wall</td>
<td>JAVA SCRIPT invariant</td>
</tr>
<tr>
<td>THETUNNEL</td>
<td>the background value must be between 0 and 20</td>
<td>JAVA SCRIPT invariant</td>
</tr>
<tr>
<td>All systems</td>
<td>Back button</td>
<td>Generic SM inv.</td>
</tr>
</tbody>
</table>
Past Executions as Oracles

Application to *Regression Testing*

1. Infer a model of a trusted version
2. Create test suite with the model
3. Run test suite on newer version
4. Check all the links and compare the states
Finding & Visualizing Relevant Differences

<table>
<thead>
<tr>
<th>Color</th>
<th>Description(s)</th>
<th>XPath original DOM</th>
<th>XPath test DOM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of child nodes</td>
<td>&lt;HTML&gt;&lt;BODY&gt;&lt;DIV&gt;&lt;DIV&gt;&lt;FORM&gt;&lt;TABLE&gt;&lt;TBODY&gt;</td>
<td>&lt;HTML&gt;&lt;BODY&gt;&lt;DIV&gt;&lt;DIV&gt;&lt;FORM&gt;&lt;TABLE&gt;&lt;TBODY&gt;</td>
</tr>
<tr>
<td>✔</td>
<td>text value</td>
<td>&lt;HTML&gt;&lt;BODY&gt;&lt;DIV&gt;&lt;DIV&gt;&lt;FORM&gt;&lt;TABLE&gt;&lt;TBODY&gt;&lt;TR&gt;&lt;TD&gt;&lt;TEXTAREA&gt;</td>
<td>&lt;HTML&gt;&lt;BODY&gt;&lt;DIV&gt;&lt;DIV&gt;&lt;FORM&gt;&lt;TABLE&gt;&lt;TBODY&gt;&lt;TR&gt;&lt;TD&gt;&lt;TEXTAREA&gt;</td>
</tr>
<tr>
<td>✔</td>
<td>element tag name</td>
<td>&lt;HTML&gt;&lt;BODY&gt;&lt;DIV&gt;&lt;DIV&gt;&lt;FORM&gt;&lt;TABLE&gt;&lt;TBODY&gt;&lt;TR&gt;&lt;TD&gt;&lt;TEXTAREA&gt;</td>
<td>&lt;HTML&gt;&lt;BODY&gt;&lt;DIV&gt;&lt;DIV&gt;&lt;FORM&gt;&lt;TABLE&gt;&lt;TBODY&gt;&lt;TR&gt;&lt;TD&gt;&lt;TEXTAREA&gt;</td>
</tr>
</tbody>
</table>

Report: Generated

Mozilla Firefox

File Edit View History Bookmarks Tools Help

file:///D:/ATUSA/report/report.html

Report: Generated

Differences (after stripped) Differences (all) Differences (all)

Compare DOMs

Done

20
Cross-Browser Testing

Step 1: Crawl web app. under different environments and capture behavioral model under each environment

Step 2: Establish pairwise equivalence of the generated models
AdSense 3.0
Example: Accessibility testing
Web Application Testing

Benefits?
- Fully automatic
- Built-in oracles
- Several types of testing

Limits?
- Nr of states
- Benefits in the trenches?

Beyond the limits?
- Scalability
- Unknown territory

Pushing the limit?
- Concurrent crawling
- State abstraction
  - Joint case studies
Part II: Testing
Plug-in Architectures

Create series of tailored products by combining, configuring, & extending plug-ins
Indigo

62 Projects
46 Million Lines of Code

http://www.eclipsesource.com/blogs/author/irbull/
One Product = Many Plug-ins
Set of Plug-ins = Many Products

Fault model:
• Interacting plug-ins,
• Plug-in configurations
• Plug-in versions
• Plug-in extensions
• Resource usage
• …

Test practices:
• Are these tested?
• Why? How?
• Why not?
Unit testing is popular

“Unit testing is where you find the most bugs”

“We don’t have UI tests, because we can cover that with unit tests”

“At least 70% of our test effort is spent on unit testing.”
Cross plug-in testing is optional

“We do cross plug-in testing bug-driven.”

“We have no automated tests for cross-plugin testing, but we do manual testing.”
Version testing is minimal

“A lot of people put version ranges in their bundle dependencies, and they say they can run with 3.3 up to version 4.0 of the platform.”

“But I’m willing to bet that 99% of the people do not test that their stuff works.”
Automated or Manual?

- **Unit Testing**: Fully automated (approx. 60%), main effort test automation (approx. 30%), main effort manual (approx. 10%), only manual testing (approx. 0%), I don't know (approx. 0%)
- **Integration Testing**: Fully automated (approx. 70%), main effort test automation (approx. 20%), main effort manual (approx. 10%), only manual testing (approx. 0%), I don't know (approx. 0%)
- **GUI Testing**: Fully automated (approx. 80%), main effort test automation (approx. 15%), main effort manual (approx. 5%), only manual testing (approx. 0%), I don't know (approx. 0%)
- **System Testing**: Fully automated (approx. 50%), main effort test automation (approx. 40%), main effort manual (approx. 10%), only manual testing (approx. 0%), I don't know (approx. 0%)
Organizational Problems Experienced

- Testing less appreciated than development activities
- Unclear who is/feels responsible for overall quality
- Restricted controllability of foreign plug-ins
- Unclear or unknown design documents or end user requirem.
- Practice too time consuming
- Unclear who is/feels responsible for performing this test practice
- No recognizable benefits of this test practice
- Test technique is perceived as less important
- Lack of time for this test practice
Technical Impediments Experienced

- Long test execution time
- Immature tooling or missing test infrastructure
- High maintenance effort
- Hard to test code tightly coupled to Eclipse
- Hard to test highly coupled, or legacy code
- Difficult to set-up test execution environment
- Lack of knowledge or expertise
Discussion

• What benefits can your technique achieve?
• Within what limits is your technique successful?

• What’s going wrong if move beyond those limits?
• *What can we do to move the limit?*
Test Automation to the Limit

Diagram:

- Main Page
  - ToDo
    - Todo List
      - Cancel
      - Save
      - Add
      - Remove
      - Yes
      - No
      - Add Item
      - Confirmation

Plugin image
More information

www.se.ewi.tudelft.nl

http://crawljax.com

http://the-eclipse-study.blogspot.com/

Twitter: @avandeursen