DOM Transactions for Testing JavaScript

Phillip Heidegger, Annette Bieniusa, and Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany
{heidegger,bieniusa,thiemann}@informatik.uni-freiburg.de

Abstract. Unit testing in the presence of side effects requires the construction of a suitable test fixture before each test run. We consider the problem of providing test fixtures for unit testing of client-side JavaScript code that manipulates its underlying web page. We propose using techniques from software transactional memory to restore the test fixture after each test run.

1 Introduction

Unit testing often requires the construction of a test fixture before running a test. As a test run may change a fixture arbitrarily, unit testing frameworks like JUnit (http://junit.org/) contain hooks for setting up and tearing down a fixture and perform each test run on a fresh instance of the class under test, tacitly assuming that the test run does not affect state outside the current instance.

Now consider unit testing for client-side JavaScript code using AJAX. AJAX code runs in the context of an HTML document. It sends asynchronous requests to a server and each response triggers a callback that dynamically modifies the HTML document. Thus, the fixture consists of the browser's internal DOM (document object model [5]) representation of the HTML document, a sophisticated network of objects with strong invariants. The code under test routinely modifies the DOM representation as well as other parts of the browser's state.

Setting up such a fixture involves running the browser to parse a slew of HTML, construct its DOM representation, render it, and run initialization scripts (e.g., onload handlers). JavaScript test frameworks either restart the browser to reset the fixture or they only offer testing on mock objects (e.g., Blue-Ridge http://github.com/relevance/blue-ridge). Neither approach is desirable, because mock objects do not offer the full functionality of the real thing and a full browser restart is too time consuming. Just consider the slowdown imposed on random testing where there are hundreds of test runs each of which takes only a tiny fraction of the time needed for a browser restart.

We propose to apply techniques from software transactional memory to the problem of restoring a test fixture. The general idea is to set up the fixture first, and then perform each test run in the body of a transaction, collect the results of the test, and rollback the transaction to restore the fixture.

To evaluate the idea, we have built an implementation based on the technique of transactional boosting [4]. To this end, our test suite intercepts calls to browser methods as well as assignments that may modify state that is shared between

```
<html><head>...</head>
<body>
 <table id="myTable">
  <tr>
   <th>Date/Time</th>
   <th>From, Subject</th>
  </tr>
 </table>
 <div id="status"></div>
</body>
</html>
```

```
function insertRow(log) {
  var tr = document.getElementById('myTable')
                   .insertRow(1);
  var td1 = tr.insertCell(0), td2 = tr.insertCell(1);
  var d = new Date();
  td1.appendChild(document.createTextNode(d));
  td2.appendChild(document.createTextNode(log));
}
function adjustStatus(str) {
  var div = document.getElementById('status');
  while (div.childNodes.length > 0)
    div.removeChild(div.childNodes[0]);
  div.appendChild(document.createTextNode(str));
}
```

Fig. 1. HTML code of the fixture and insert row into table and adjust status.

```
while (runAnotherTest()) {
   txn.init();
   var data = tests.generateTestData();
   var res = insertRow(data); // run transformed test case
   tests.processResult(res); // check and store result
   txn.rollback();
}
```

Fig. 2. Random testing of insertRow within transactions.

runs. For each modification, a compensating action that reverses the effect of the modification is pushed on an *undo stack*. When a test run is finished, the compensating actions are executed in a LIFO manner such that their total effect is to rollback the global state (including the DOM) to its original state.

2 Motivating Example

In the context of an AJAX project, consider the task of keeping up-to-date a table of incoming emails and a status line that contains the number of unread emails in the inbox (Fig. 1 left). To this end, the project contains some JavaScript code that periodically contacts a server for the information and then asynchronously updates the HTML document in a callback. The functions insertRow and adjustStatus (Fig. 1 right) are invoked from the callback. The former extends the table with a new line whereas the latter changes the status text.

Suppose we want to find bugs in the function insertRow using random testing with the fixture described by Fig. 1 (left). For each test run, the testing framework first has to generate a random string, then run the function against the fixture, and afterwards use DOM functions to check if the expected change has been performed on the document structure.

To use the same fixture for each test run, our test framework rewrites the code of insertRow and adjustStatus by inserting code that registers actions for all side

effects. The resulting code for a test run needs to be wrapped in a transaction as shown in Fig. 2. Before executing the test suite, starting the browser and loading the program under test creates the fixture (the initial HTML page). txn.init marks the start of a transaction. During the test run, the compensating actions are transparently pushed on an undo stack. After the test run, txn.rollback triggers the processing of the undo log that restores the initial state.

3 Implementing a transactional layer in JavaScript

To perform test runs in a transaction, it is necessary to rewrite the code under test such that each operation with global side effects registers a compensating action. Such side effects are assignments to global variables and properties of globally reachable objects as well as calls to native DOM methods that modify the DOM. Unfortunately, JavaScript's `with` statement prevents a static analysis of the scope of a variable in general. Similarly, in many cases it is not possible to statically decide which method is called. Thus, the rewriting implements the following strategy:

- For each assignment, push a closure on the undo stack that assigns the old value to the variable or object property.
- For a method call, the decision whether a user method or a native DOM method is invoked is taken dynamically in a library method, which checks the method's closure against the known DOM methods. For each DOM method, the library provides a factory to create a compensating operation.[1]
  For a user method, no compensation is needed because each side effect inside of the user method creates its own entry in the undo stack.

Under a closed world assumption where no additional code is loaded at run time or generated using `eval`, this transformation is safe in the sense that it does not change the semantics of the code under test and that it faithfully registers compensating actions for all relevant side effects.

We integrated the transformation in our tool JSConTest[3][2], a random test generator for JavaScript based on contract annotations. To increase performance of the test suite, the annotation `~noEffects` informs the transformation that the function is free from side effects.

4 Related Work

Dhawan and co-authors [1] propose to augment the language and runtime of JavaScript with transactions for monitoring security policies. In contrast, we transform the code under test. Hence, the tool can be used in all browsers which implement the current ECMA Script standard.

[1] So far, we were able to provide compensating actions for all methods of the DOM API that we came across.

[2] http://proglang.informatik.uni-freiburg.de/jscontest/

Checkpointing [2] is a related technique that creates recovery points for long running software to avoid restarting such applications from scratch. In browsers which offer the possibility to construct a checkpoint and revert to it, the rollback of a fixture could be performed without administration of transactions. It has also been used to construct debuggers that support backward evaluation [6].

There are also proposals for testing programs involving databases using transactions. Again, the rationale is to avoid the cost of recreating an expensive fixture between test runs.

5 Conclusion and Outlook

We have shown how transactions enable the construction of an effective JavaScript testing framework that works in the original browser environment and avoids the repeated reconstruction of test fixtures / HTML pages from scratch. This approach simplifies the setup of a AJAX testing environment enormously and allows for testing under real-world conditions.. We have constructed a proof-of-concept implementation as an extension of our tool JSConTest[3].

The implementation can be further improved in several ways. Due to their irreversible nature, I/O operations like sending HTTP requests have to be treated differently. For example, if a testing environment includes a server, this server has to be reset for each run. Similarly, methods on the `window` object should be redirected to a mock object to avoid human interactivity during the tests.

Compensating actions should be inverses of the original actions. This property should be (mechanically) proved.

Program analysis can improve the performance of the rewritten code. In ongoing work, we explore the partial inference of scope information so that local and global variables can be distinguished outside of `with` statements.

References

1. M. Dhawan, C. Shan, and V. Ganapathy. The case for JavaScript transactions. In *PLAS'10: Proceedings of the ACM SIGPLAN Fifth Workshop on Programming Languages and Analysis for Security*, Toronto, Canada, June 2010. ACM Press, New York, NY, USA.
2. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
3. P. Heidegger and P. Thiemann. Contract-driven testing of JavaScript code. In *TOOLS*, Malaga, Spain, June 2010. Springer. To appear.
4. M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.
5. P. Le Hégaret, R. Whitmer, and L. Wood. W3C document object model. http://www.w3.org/DOM/, Aug. 2003.
6. A. Tolmach and A. W. Appel. A debugger for Standard ML. *J. Funct. Program.*, 5(2):155–200, 1995.