Automatic Invariant Finding in Dynamic Web Applications

Frank Groeneveld

Automatic Invariant Finding in Dynamic Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Frank Groeneveld
born in Rotterdam, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
ewi.tudelft.nl

Patrijsweg 80
2289 EX Rijswijk
the Netherlands
tamtam.nl

Automatic Invariant Finding in Dynamic Web Applications

Author: Frank Groeneveld
Student id: 1262998
Email: frankgroeneveld@gmail.com

Abstract

Web applications are rapidly becoming more advanced since the introduction of AJAX technologies. Famous examples include Google's GMail, Maps and Docs, as well as Twitter and Facebook. These technological advancements bring along a number of challenges, mostly concerning web application testing. In this thesis, we propose a number of techniques to automatically test web applications using invariants on the application's Document Object Model (DOM) and invariants on the application's JavaScript code. Using a proxy, we add JavaScript source code that can generate an execution trace. This trace contains all information about the JavaScript variables that is needed to derive invariants on them. Next, we crawl the web application. This can be done manually or in an automated fashion. Crawling the application will generate the actual execution trace. The invariants can be derived using special tools that analyze the trace. Furthermore, during crawling, the DOMs of the web application are fed to an algorithm that can derive invariants on the DOM. We implemented all this as plugins to Crawljax, however these techniques are not only appliable to Crawljax. To evaluate the quality of the invariants, we conducted several case studies. The results are encouraging, meaning our work can be used to automatically generate invariants which can be used for regression testing of dynamic web applications.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCMS, TU Delft
University supervisor:	Dr. Ir. A. Mesbah, Faculty EEMCMS, TU Delft
Company supervisor:	J. Hulscher, Tam Tam
Committee Members:	Dr. M.M. de Weerdt, Faculty EEMCMS, TU Delft
	Dr. A.E. Zaidman, Faculty EEMCMS, TU Delft

Preface

When I first visited Arie van Deursen about a year ago, I initially had the idea to do "something with AJAX and testing". Arie started to laugh and introduced me to Ali Mesbah and his PhD project: the Crawljax web application crawling and testing tool. Ali's work turned out to be exactly what I was looking for, because Arie and Ali were interested in extending Crawljax with automatic testing capabilities. I had the possibility to develop this extension at Tam Tam in Rijswijk.

After nine months of work at Tam Tam, developing Pleaserobme[1] as a side-project with two colleagues, numerous meetings and brainstorm sessions with Ali and a visit to Google London, I have finally finished my master's thesis.

I would not have finished this project successfully without the help of a number of people. First of all, I would like to thank Ali for all his help and support, I learned a lot from him. Also he tried to make me think in opportunities instead of problems, which is something for me to keep in mind. Secondly, I want to thank the people at Tam Tam in the "adviesgroep" Het Nieuwe Samenwerken because of their support. It was a really pleasant group to be part of. Finally, I would like to thank my parents, my brother and my girlfriend for supporting me throughout the process. Although I explained what my master's project was about a couple of times, they are still not able to explain it to others.

Frank Groeneveld
Rijswijk, The Netherlands
28th of June 2010

[1] `http://pleaserobme.com/`

Contents

List of Figures

Listings

Chapter 1

Introduction

In this chapter we will introduce various existing tools and testing techniques. Also, we will introduce our problem description.

1.1 Invariant-based Testing

Since the beginning of computer programming, researchers have been trying to find good testing procedures. One of those testing procedures is using invariants. A program invariant is an expression defined over variables of an algorithm or software program that should evaluate to true on every function entry or exit point [19]. Consider the simple Java example shown in listing 1.1.

Listing 1.1: Invariant example.

```
public class MonthYear {
    private int month;
    private int year;

    public MonthYear(int m, int y) {
        month = m;
        year = y;
        assert invariant();
    }

    public changeMonth(int m) {
        assert invariant();
        month = m;
        assert invariant();
    }

    private boolean invariant() {
        if (month <= 0 || month >= 13) {
            return false;
        }
        if (year < 0) {
            return false;
        }
        return true;
    }
}
```

This example shows a class that can hold a month and a year. The class has a method named `invariant` that checks whether the year is a positive integer and whether the month is an integer between zero and thirteen. At the end of the class constructor this invariant is tested by using the `assert` method. If the expression that follows the assert fails, the Java virtual machine (JVM) will throw an exception and the execution will stop (when the exception is not caught).

Note that this method requires the programmer to insert assert statements in all functions of the class to ensure the invariant still holds. A number of extensions to Java have been developed to automatically check invariants on classes, for example the Java Modeling Language (JML).[1] If we rewrite the MonthYear example of the introduction using JML, we get the code that is shown in listing 1.2.

Listing 1.2: JML invariant example.

```
public class MonthYear {
    private int month;
    private int year;

    /*@ invariant month >= 1 && month <= 12 @*/
    /*@ invariant year >= 0 @*/

    public MonthYear(int m, int y) {
        month = m;
        year = y;
    }

    public changeMonth(int m) {
        month = m;
    }
}
```

As can be seen above, the invariants are defined in a special type of comments: annotations. The programmer does not have to insert assertions in every class function anymore, this is automatically done by the JML preprocessor.

Since the beginning of software development, Alan Turing has been advocating the use of assertions and invariants while writing algorithms. In 1949 he wrote [30]:

> In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.

So according to him assertions are not only there to check program validity, but also to give programmers a better understanding of their program. Turing even used them to prove the correctness of a program.

1.2 Web-based Testing

The web has evolved in numerous ways since its first incarnation in 1989. One of the latest developments is the use of Asynchronous JavaScript and XML (AJAX) [9]. This

[1] `http://jmlspec.org/`

new method of browsing the web makes a lot of old techniques, for example testing tools, obsolete. Mostly this is because they rely on the fact that the URL of a page can be used as a unique identifier and state access mechanism. In order to cope with these problems, Crawljax[2] was developed [17]. Crawljax is a tool that makes it possible to crawl AJAX web applications while inferring a state machine. This state machine can then, for example, be used to create a static mirror of the web application.

By combining Crawljax and the invariant testing technique [18], one is able to define invariants on the Document Object Model (DOM). These invariants can be used for instance to test whether the DOM is valid HTML.

1.3 Problem Description

In this master thesis we try to combine invariant based testing with web-based testing, since highly dynamic web applications are difficult to test [16]. We try to automatically find and derive invariants in web applications in order to improve the quality of web applications without increasing development costs much.

In order to do this, we define the following research questions.

- How can we automatically find invariants on JavaScript variables of web applications? How can they be used for testing?
- How can we automatically find invariants on the DOM of web applications? How can they be used for testing?
- How effective are these automatically found invariants when used for testing?

[2] `http://crawljax.com/`

Chapter 2

Related Work

In this chapter we will have a look at various testing and analysis tools in both web applications, as well as traditional applications.

2.1 Invariant Derivation

Finding invariants manually for big software projects can take up quite some development time, especially if the time between writing the invariants and developing the algorithms increases. Therefore, it would be useful to have a tool to automatically detect invariants. Extensive research has been done to create such tools, both in the commercial world as well as in the academic world.

Daikon[1] is an open source automatic invariant finding tool that was developed by researchers at Massachusetts Institute of Technology. It can be used to automatically derive invariants for certain programming languages and can be downloaded for free. Daikon can be used to detect a variety of different invariants [8].

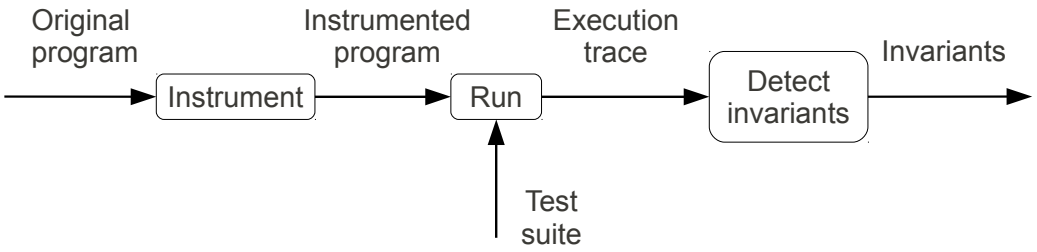

Figure 2.1: Architecture of the Daikon tool for dynamic invariant inference [6].

The basic architecture of Daikon is shown in figure 2.1. Daikon works as follows. The original program is instrumented by Daikon to support trace creation. Next, the instrumented program is run using a test suite and the data trace that is generated is stored somewhere on disk. This trace is then used by Daikon to derive invariants for the program.

[1] `http://groups.csail.mit.edu/pag/daikon/`

If it is possible to generate a trace for a program, then it is possible to automatically find invariants using Daikon. A trace is basically a huge log of *(value, variable)* pairs that were encountered while running the program. Most of the time, traces are created using test suites, as can be seen in figure 2.1.

Using a test suite means the quality of the invariants found by Daikon, depends on the quality of the test suite. Test suites can only prove non-quality [20] and are therefore difficult to produce. However, in practice Daikon seems to perform well even if modest test suites are used [24].

Daikon uses a method based on brute force invariant finding. The developers of Daikon created a list of invariant patterns and before reading any of the traces, Daikon assumes all these invariants are true. When an invalidating example is found in the trace, the invariant is removed.

Because there are lots of invariants for variables possible, this method becomes rather cumbersome if the number of variables increases. Therefore Daikon uses various techniques to improve this brute force method in order to make it more scalable [7].

Agitator[2] is a commercial tool developed by Agitar Technologies. The developers of Agitator leveraged several test automation techniques from the academic world, including Ernsts work on Daikon, and integrated those techniques in what they call software agitation [4]. Agitator is a software agitation tool that helps developers with (unit) testing their own code.

Agitator integrates test data generation with automatic invariant detection. Test data generation is the first step in the agitation process. The test data is generated using static and dynamic analysis of Java code at the byte-code level. In order to satisfy performance requirements, some approximations and heuristics are used. When the test data generation phase is finished, this data is used as a kind of test suite.

The second step in the agitation process is running the test suite to produce a trace of the program. Eventually, this trace is used to derive invariants for the code using an algorithm that works much the same as Daikon, though it was developed independently [4]. Agitators invariant deriving algorithm is based on the *simple incremental algorithm* described by Perkins and Ernst [24], but uses different optimizations and heuristics in order to make the algorithm scalable.

Furthermore, the invariant derivation algorithm preserves implied invariants. This is done because the Agitator tool is used interactively and reducing the number invariants using implication is proved to be bad for response times. Also it might hide bugs that the developer can only see if all invariants are stated, instead of just the reduced ones.

DIDUCE[3] is a free, open source tool that aids programmers in finding complex software bugs and identifying their root causes [12]. It was developed at Stanford University. DIDUCE instruments a program to allow dynamic invariant detection while running the program. During this instrumented run, the invariants are relaxed on every invariant violation and all these violations are reported to the programmer.

DIDUCE has two modes of operation: training mode and checking mode. In training

[2] `http://www.agitar.com/solutions/products/software_agitation.html`

[3] `http://diduce.sourceforge.net/`

mode, DIDUCE silently learns about variables and their possible values. Invariants are deduced from these values while the program is running, but any relaxations of the invariants are not reported. However, the checking mode does report relaxations found during execution. It is this checking mode that can be used to find bugs and their possible causes.

DIDUCE instruments static program points to check values of tracked expressions, report invariant violations and update invariants. This is all done directly on Java byte-code, which means DIDUCE does not need access to the source code.

Another invariant derivation tool is *DySy* [5]. The developers of DySy wanted to create a better method for deriving invariants dynamically. They developed an algorithm that can perform a symbolic execution of the program simultaneously with its concrete execution. DySy is not a commercial product, but it is not available for download either.

Symbolic execution can be seen as an enhanced testing technique, that executes a program symbolically for a class of inputs. Instead of executing a program on a set of real inputs, it is executed on a set of "symbols", which means the results are symbolic formulas over the input symbols [15]. For example, take a look at the `square` function in listing 2.1.

Listing 2.1: Sample square function.

```
function square(x) {
    return x*x;
}
```

When this function is tested using normal inputs, the results that are found are the squares of these inputs, for example, input 2 gives result 4. However, using symbolic execution on symbol a, the result is a^2. For advanced functions, symbolic execution can therefore give the programmer insights that were difficult to derive manually.

DySy uses symbolic execution to obtain invariants, preconditions and postconditions that are not based on predefined patterns (unlike Daikon, which has a huge list of "invariant patterns"), but on expressions that are encountered by the symbolic program execution.

DySy was implemented using Pex,[4] a dynamic analysis and test generation framework for .NET. Pex monitors the execution of a program by instrumenting the code. This instrumented code drives a "shadow interpreter" in parallel with the actual program execution. Every .NET instruction does a callback to Pex, which then executes the operation symbolically.

The interpreter records all branch conditions that are executed as well as implicit branches. Implicit branches are implicit checks done by the .NET runtime environment, for example following a reference is only done if the reference is not `null`.

DySy starts an interpreter when a method call is done. The interpreter stops when that method call finishes. Next, DySy uses the branching conditions that were recorded by the symbolic execution to compute path conditions and final states of the method. These are then used to compute invariants, preconditions and postconditions. First, the invariants are found by looking for terms that appear in both the path conditions and in the final states of the method. Using the invariants found, it is now possible to reduce the path conditions and report their disjunction as precondition for the method. The postconditions

[4] `http://research.microsoft.com/Pex/`

of a method are computed using the conjunction of all path-specific postconditions. A path-specific postcondition is an implication with a path-condition on the left hand side and a conjunction of equalities on the right hand side. For example, if a method sets *result* to 1 if both argument *a* and argument *b* are positive, the post condition becomes:

$a > 0 \wedge b > 0 \rightarrow result = 1$

ClearView is a system that can automatically patch errors in deployed software [25]. It can do this without the need for source code access or access to debugging information. The algorithm used to achieve this consists of various steps.

To evaluate the effectiveness of ClearView, DARPA[5] hired Sparta Inc.[6] to do a so-called Red Team exercise. A Red Team exercise can best be described as a security audit done by an external, independent team. Sparta used several exploits for the Firefox web browser and tried to execute arbitrary (attacker-chosen) code. With the unprotected version of Firefox, they were able to do so with all of their exploits. ClearView was able to detect and block all attacks, terminating Firefox before harmful code could be executed. Furthermore, ClearView generated patches for seven of the attacks.

2.2 Testing Web Applications

2.2.1 Traditional Testing

VeriWeb is a tool that can automatically discover and systematically explore website execution paths that can be followed by a user [2]. VeriWeb supports form submission, which can be configured using so-called SmartProfiles. Furthermore, VeriWeb can be used to detect errors in navigation and pages by use of plugins.

ReWeb is another "traditional" web testing tool, which tries to generate basic UML models of web applications [28]. The models can be used along with coverage criteria to write test-cases. Andrews et al. [1] try to achieve the same thing using a finite state machine and user defined path conditions between the states.

2.2.2 Web 2.0 Testing

The use of AJAX in web applications results in a much better and richer user interface. However, the use of AJAX also includes some challenges. Most (if not all) search engines are not able to index data that is normally retrieved through AJAX. This means that websites that use AJAX technology will be difficult or impossible to find through search engine. Furthermore the testability of AJAX websites becomes even more difficult, because most existing tools are based on unique URLs that could be used as identifiers for states, but this is not the case for AJAX websites.

Currently, several testing tools that can cope with these new challenges exist. Some of them, for example Selenium,[7] are based on a capture-and-replay technique. These tools record actions that a user takes in a browser and are able to replay them later. Other tools,

[5] `http://darpa.mil/`

[6] `http://sparta.com/`

[7] `http://seleniumhq.org/`

such as Webdriver,[8] allow the developer to programmatically control the web browser, thus allowing rich interaction with AJAX web applications.

Crawljax is a tool built on top of browser controlling techniques. It exercises client-side code and fires events on clickable elements that change the DOM using a real web browser. All states that are found using this technique are added to a graph. Eventually, the graph will contain all user interface states and the possible transitions between them.

Crawljax uses a dynamic approach to discover possible states and the corresponding DOM trees because a static approach seems impossible to implement. Crawljax works as follows [17]. When Crawljax is initiated it will start a browser and load the specified URL. The state machine is then initialized to only contain this first state: the web page as it is right after loading the page. The next step in the crawling process is the determination of candidate clickables, elements that might change the DOM after being clicked at. Crawljax then clicks one of those elements. After the click event, the DOM is compared with the previous DOM (the one from before the click event). Crawljax has found a new state when these two DOMs differ from each other and all previously found DOMs. When the new state has been added to the state machine, the process creates a new list of candidate clickables and continues its search.

Quality demands for web applications seem to constantly rise, thus testing AJAX web applications becomes more important. In previous tools, the user still has to "record" his actions and then has the ability to replay them. The method used by Crawljax is fully automatic.

Because Crawljax has access to all the different dynamically created DOMs, these DOMs can be checked against pre-defined invariants during the crawling phase. There are currently a few possible invariants that can be defined [18]:

- Generic DOM invariants. For example, these invariants make sure the DOM is valid and does not contain any error messages.
- State machine invariants. For example, to make sure that there are no dead clickables and the back button works correctly.
- Application-specific invariants. For example, user-defined invariants over the DOM-tree to check whether certain elements are always visible.

Furthermore, Crawljax has a rich plugin Application Programming Interface (API), which allows developers to create plugins that test the application while it is being crawled. For example, a plugin can check whether the front-end of a web application represents the state of the back-end correctly.

DoDOM is an automated system for detecting errors in AJAX applications [23]. This is done using dynamic analysis during the execution of a given sequence of user actions. During the execution, DoDOM observes the behaviour of the application and identifies possible invariants over the applications DOM structure.

The algorithm that is used to find invariants starts by using the first DOM as its *invariant DOM*, meaning that everything in that DOM should be available in all states. Next, it

[8] `http://code.google.com/p/webdriver/`

executes a user action and checks for changes between the nodes in the invariant DOM and the nodes in the current DOM. If any of the following changes are found, the node is removed from the invariant DOM:

- Contents of the nodes differ from each other.
- A node in the invariant tree has more children than in the current DOM.
- The invariant tree has nodes that are not found in the current DOM.

After executing all user actions, the invariant DOM is finished and can be used to test the application. Most of this algorithm is implemented in JavaScript, hence, some problems arise when detecting certain DOM modifications.

DoDOM was evaluated by doing numerous er of tests on real world web applications. These experiments showed that DoDOM was able to detect the following errors:

- Events which affect the DOM were dropped.
- Domains that serve content being down or return errors.
- Faults that have impact on future events.

2.3 Web Application Analysis

A number of scientific techniques have also been applied to web applications for testing and analysis. A number of the tools that were created out of this will be introduced in the following sections.

Kudzu is a symbolic execution system for JavaScript that can do automated vulnerability analysis [29]. This is done by automatically generating a high-coverage test suite using symbolic execution of the JavaScript source code. This test suite can then be used to search for client-side code injection vulnerabilities.

Initial evaluations of Kudzu seem promising. During the evaluation, Kudzu found eleven vulnerabilities, of which two were previously unknown. Furthermore, none of the vulnerabilities were false positives.

BrowserShield is a framework that can perform dynamic instrumentation of JavaScript to do vulnerability driven filtering [27]. Instrumentation of the JavaScript code is done at a proxy or firewall, so deployment of the tools is simple.

The actual vulnerability filtering is done using so-called policies. These policies are basic JavaScript functions that can, for example, filter out certain vulnerable ActiveX objects to stop them from being constructed. Using this technique, BrowserShield allows patching of all Microsoft Internet Explorer vulnerabilities that were targeted by BrowserShield targeted, meaning the browser can be protected 100% at the network level. This allows system administrators to protect their users before patches are rolled out.

AjaxScope is a dynamic instrumentation platform that enables cross-user monitoring and just-in-time control of web application behaviour on end-user desktops [13]. AjaxScope does this by modifying (instrumenting) JavaScript that passes a proxy server. Part of

the website visitors receive instrumented JavaScript that check, for example, whether any endless loops or memory leaks occur during execution. Results of these verifications are sent back to the proxy server which can then save them in a log file.

AjaxScope demonstrates the concept of *instant redeployment*: the ability to serve different versions of the web application to individual users. Using instant redeployment, some users can test new features or fixes. The AjaxScope platform provides the infrastructure to achieve this in a simple manner.

Chapter 3

JavaScript Invariants

In this chapter we will explain how JavaScript invariants can automatically be detected and how they can be used for testing.

3.1 Deriving Invariants

In general, invariant derivation is done using a workflow that looks similar to figure 2.1 which is composed of the following steps:

1. Find a way to log variable values during program execution, to obtain an execution trace.

2. Execute the program (manually or in an automated fashion).

3. Derive possible invariants using the log that was produced in the previous step.

This workflow can also be used to derive JavaScript invariants. In the following sections we will explain how.

3.1.1 Obtaining an Execution Trace

The first step, finding a way to log variables and their values during program execution, can be done using two different techniques. Either by modifying the JavaScript code to log the variable values or by modifying the JavaScript runtime to produce the log for us. Both of these techniques have their own trade-offs as will be demonstrated in the following sections.

Modifying the JavaScript Code

Modifying the JavaScript code to log variable values can be done manually or automatically using a parser. Both of these methods add so-called instrumentation code [10] to the program. The instrumentation code creates a log containing the actual values of the variables, for example by saving them in a file.

Algorithm 1 Automatically adding instrumentation code.

```
1: {parse source code to an AST}
2: AST ← parseSource(source)
3: {walk the AST}
4: for node ∈ AST do
5:     if node instanceof FunctionNode then
6:         {get function body}
7:         body ← node.getBody()
8:         {add instrumentation code as first line of function body}
9:         body.prepend(instrCode)
10:        {walk the function body}
11:        for line ∈ body do
12:            if line instanceof ReturnNode then
13:                {add instrumentation code before returning from the function}
14:                body.prepend(line, instrCod)
15:            end if
16:            if isDOMModification(line) then
17:                {add instrumentation code before and after the DOM modification line}
18:                body.prepend(line, instrCod)
19:                body.append(line, instrCod)
20:            end if
21:        end for
22:        {set the new body}
23:        body.setBody(body)
24:    end if
25: end for
26: {generate plain source code}
27: source ← AST.toSource()
```

The biggest problem encountered when manually adding instrumentation code is the fact that it needs maintenance and is expensive to create. If, for example, a function is modified to require an extra argument, all the instrumentation code in that function needs to be modified to also log that argument. Furthermore, it is not advisable to keep the instrumentation code in the production version of the script, because of the increased file size and the performance hit. This means that you would have to manually update two versions of the script: the production version and the instrumented development version. All in all, this seems like the worst possible option.

Automatically adding instrumentation code solves the maintenance problem, because the instrumented version can automatically be produced from the normal production version. Using a parser it seems relatively easy to produce an Abstract Syntax Tree [11] (AST) and traverse it while adding instrumentation code where needed. After the traversal, the AST can be converted back to JavaScript source code again. An example of such an implementation is shown in algorithm 1.

First, the source code is parsed into an AST, which is then traversed using a for loop. When a *function definition* is found, the first statement of the function body is prepended with instrumentation code, which can save all variables and their values that are currently in scope. Next, the code searches the function body for return statements to prepend these with instrumentation code as well. This is done because we are interested in finding invariants for these program points, they can be used as pre and post conditions.

Automatically adding instrumentation code can even be done on the fly [13, 14], using a modified proxy server. The proxy server can capture all JavaScript that comes by, add the

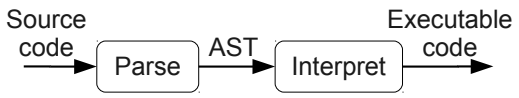

Figure 3.1: Source code interpretation.

instrumentation code and forward it to the client.

One of the most important problems encountered when trying to log variable values using code instrumentation is the fact that JavaScript does not support writing to files (at least not yet, a specification has been proposed [26]), so the instrumentation code cannot write the variable values to a file.

Although this last problem seems difficult to tackle, it is useful to do so, because a tool that can automatically instrument JavaScript code would not need a lot of maintenance or updates. After all, the syntax of JavaScript is not something that changes frequently, so the tool could probably be used for a long time without modifications.

Saving the Execution Trace

Saving the execution trace can be done using two different methods. First of all, it is possible for the instrumentation code to send the log to the proxy server or the web server using an AJAX request [13]. The server or proxy can then save it to a file or store it in a database. Secondly, we could store the log somewhere in the browser and use another application to read it out. We could, for example, save it in a JavaScript array. A combination of these two methods should also be considered: buffer a number of log requests and send them to the proxy server once in a while.

Modifying the JavaScript Engine

The JavaScript engine is the part of a browser that normally executes the JavaScript when a user navigates to a website. The most popular browsers all use their own implementation of a JavaScript engine.

Modifying the JavaScript engine to log variable values at certain program points is another possible way to obtain an execution trace. The variable logging can be done in two stages of the engine execution. Either in the parsing stage or in the interpreting stage (see figure 3.1).

At the parsing stage, it is not really possible to log variable values, because nothing is evaluated at that stage yet. Therefore, this stage can only be used to add instrumentation code, similar to the previous section, to the JavaScript code while it is converted to an AST. The interpreter step would not need modifications using this method.

Logging at the interpreting phase also uses an AST, because an interpreter basically just traverses an AST (in a certain order) and "executes the meaning" of the nodes it visits. The modified interpreter would not only execute the nodes, but at certain nodes (program points) it would create a dump of all variable values. Some JavaScript interpreters, like We-

bkits SquirrelFish,[1] use bytecode instead of an AST for performance reasons. A bytecode interpreter still has the possibility to recognize the program points we are interested in, so that would not introduce any problems.

One of our main concerns when modifying the JavaScript engine is the fact that it is browser dependent, because most browsers have their own JavaScript engine implementation. This means that we would need to modify the JavaScript engine of every browser we would like to use. Adding variable logging code also adds a performance hit, even when using an implementation that can be switched off using a boolean variable, because it would still result in many checks whether to log variable values or not. Furthermore, JavaScript engines evolve pretty fast, so keeping the changes working with newer versions of the engine might cost a lot of time.

3.1.2 Execution of the Program

Step two of the invariant derivation workflow consists of executing the instrumented program. This is done in order to log the variables to an execution trace. For JavaScript this means the web application needs to be browsed in a browser. The execution should try to run as much of the JavaScript code as possible and execute it in different ways, for example with all kinds of values for the arguments of functions.

This can be done manually, but to find good invariants it is better to do a very extensive execution, which would be too expensive to develop by hand. Therefore automatic execution seems to be the best option. One can use a test-suite, if available, that uses browser controlling libraries such as Webdriver,[2] Watir,[3] or Watij.[4]

There are also a number of automatic execution tools available, most of them based on a "capture and replay" technique. Initially, the user clicks through the web application and all these actions are recorded as a macro. This macro can then be replayed endlessly. However, Crawljax can actually crawl AJAX web applications automatically. Tools such as Crawljax seem like the best option to use, because they allow for almost completely automatic execution of the JavaScript code.

3.1.3 Contents of the Execution Trace

Up until now we have not talked about the contents of the execution trace we are trying to obtain. There are two interesting cases we can instrument. We will explain them in the following sections.

JavaScript Variables

Our first interest is of course the values of the JavaScript variables, because we can use these values to derive invariants on the variables. We include *global variables*, *function arguments* and *local variables* in the trace. The values are logged at function entry and

[1] `http://webkit.org/blog/189/announcing-squirrelfish/`
[2] `http://code.google.com/p/selenium/`
[3] `http://watir.com/`
[4] `http://watij.com/`

function exit points. Function entry is a simple case: logging has to be done at the first line of the function. Function exit means the last line of the function, but also the line before any intermediate return statements that occur in the function body.

By using a certain format for the program point names, we can also add information about the *script name*, the *function name* and the *line number*. This information can be useful when we find a failing invariant during the testing of the application, because it can guide the developer to the possible bug.

The information about the variables in scope consists of their *names*, *runtime types* and *values*. The runtime type is stored because JavaScript is a loosely typed language, i.e. you cannot derive the types of variables syntactically, so we need to discover the variable types at runtime.

Dynamic DOM Modifications

The other interesting case we want to include in the execution trace is the modification of the DOM using JavaScript functions. These kind of modifications are interesting because they can give a better understanding of highly-dynamic web applications. Also, these are the kind of web applications that are usually difficult to test [16], so finding invariants over them can be really useful.

Most of the time, DOM modifications are done in a certain "pattern". This pattern can be detected and we can add instrumentation code before and after it occurs. An example pattern is shown in listing 3.1. First, some function or framework is used to "select" or find the element(s). Next, a function is called on the object that was returned. This function does the actual modification of the DOM.

Instrumentation of such a pattern seems simple: search for certain function calls that are done on objects and add instrumentation code before and after the pattern occurs. However, some frameworks such as jQuery[5] have functions that sometimes read DOM values and sometimes set DOM values depending on the number of arguments. For example, the jQuery `attr` function can be used to read an attribute if it is used as follows:

```
jQuery('body').attr('id');
```

But it sets the attribute value when used in this way:

```
jQuery('body').attr('id', 'something');
```

Recognizing the DOM modification patterns therefore requires more knowledge than just the function name. After conducting some experiments, we came to the conclusion that checking the name and the number of arguments seems sufficient to recognize DOM modifications (for jQuery at least).

Listing 3.1: DOM modification using JavaScript.

```
/* select the elements we're interested in */
var elements = jQuery('ul li:last-child');
/* add 'last' to the class attribute of the elements */
elements.addClass('last');
```

[5] `http://jquery.com/`

3.1.4 Invariant Derivation

The final step for invariant derivation is the actual derivation itself. Some extensive research has been done on this subject, as can be read in section 2.1. We will adept these existing techniques for JavaScript.

Most of these techniques are based on the brute force method: for all variables, consider all possible invariants to be true. Iterate over the list of found values and remove any invariant that fails with these values. The brute force method can be optimized in a number of ways [24].

For our work, we adapted Daikon. We extended Daikon with support for generating output in JavaScript syntax. This means the invariants that are found can be tested directly by running them in a JavaScript engine.

In this final step, we execute Daikon over the execution trace we obtained earlier. Daikon can then derive possible invariants. All output produced by Daikon is saved in a file. In the following section, we will use this file to test the web application.

3.2 Testing Invariants

After finding JavaScript invariants, we need a way to use them, for example to test for regressions. Using these invariants is more difficult than one might think.

Initially we had the idea of using Crawljax to test the invariants we found. Generally speaking, Crawljax can only access global JavaScript variables. This is done by executing a piece of JavaScript in the browser and sending the results back to Crawljax. This means that we have "programming level access" to the variables, we cannot access them using some mechanism that reads the internals of the JavaScript runtime.

From this "programming level access" it is not possible to access local variables of functions. It is still possible to access global variables and these seem to be the most useful variables for invariants. Invariants should always evaluate to true, so we should be able to test them at any given time. However, JavaScript has a feature that makes it difficult to find an application with global variables in the first place: it supports closures.

> A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression) [22].

Nowadays, a lot of JavaScript code found on the internet is wrapped in a closure in order to avoid collision of variable names for example. This means it is not possible to access the variables from somewhere else in the JavaScript program.

In order to solve this problem, we moved the invariant testing to a different level: we inserted it in the JavaScript source code using a similar method as adding the instrumentation code to find the invariants, namely by generating an AST in the proxy and modifying that AST. By adding invariant tests between the normal statements, the invariant testing code has access to all the variables it needs and it can still save the results in a globally accessible array. This globally accessible array can then be read by a Crawljax plugin which can generate a testing report.

This new approach allows us to do even more testing. We can now test preconditions and postconditions of methods by inserting the correct code at the beginning or end of the method. So the problem we encountered brought us a better solution in the end.

3.2.1 Inserting Found Invariants

Because the invariants are found for variables at certain program points, they must be true at those program points. This implies that we can only test whether these invariants are true by running the tests at the same program point as where they were found. In practice, this means that we have to use the same techniques as the ones we used for logging the variable values: either automatically add the testing code to the JavaScript source code, or test the invariants in the JavaScript interpreter. The same problems and advantages apply here for both as well.

An example of JavaScript code that can be used to test whether an invariant is true or false is shown in listing 3.2.

Listing 3.2: JavaScript assertion code.

```
/* check whether the invariant is true */
if(!(first > 2 && first < 7)) {
    /* not true, add an entry to the assertionFailure list */

    var entry = new Array();

    /* scriptname, functionname, line number */
    entry.push('scriptname.functionname.21');

    /* invariant that failed */
    entry.push('first > 2 && first < 7');

    window.assertionFailures.push(entry);
}
```

Automatic insertion of the invariants in the JavaScript code is a browser independent way of testing them and will not need much maintenance or updates. The output produced by Daikon contains the program point names and the corresponding invariants. We can derive the position of the program point from its name. For example, a program point might be named: `http://test.com/script1.calculate:::ENTER` This means the program point is positioned at function entry of the calculate function, which can be found in the first script tag of the HTML page that can be found at `http://test.com/`. This means we can always find out which invariants should be tested at which program points.

3.2.2 Executing the Tests

After adding the testing code to either the JavaScript source code itself or the JavaScript runtime, we can check for regressions by executing the program again. This can be done by using the same test suite or crawling tool that was used to find the invariants in the first place. When the test run is finished, the results can be used to debug errors because they

can give precise information about where the invariants failed, including the filename, line number and function name of the program point.

Chapter 4

Automatically Finding DOM Invariants

Web applications have another component which is not thoroughly tested when we focus on JavaScript invariants, namely the Document Object Model (DOM). The DOM represents the elements of an XML or HTML document by placing them in a tree-like structure. This means the DOM for certain web applications can contain invariants as well. An example could be the existence of a `DIV` element with an `id` attribute containing the string ”menu”. A user can get confused if the menu is missing on some pages, so checking whether this invariant holds for all pages can be considered a useful thing to do.

In this chapter we will explain how DOM invariants can automatically be detected and how they can be used for testing.

4.1 Deriving Invariants for One State

Initially we focused on developing an invariant derivation algorithm that could derive invariants for a certain given DOM. This algorithm turned out to be very simple. For every element in the DOM, we generate an XPath expression that describes the element, its attributes and its attribute values. The XPaths are stored in such a way that the parents and children of the element can also be derived. We currently use indentation for this, because it gives us a simple and clean format. We call all XPath expressions, which describe the invariants, the *invariant DOM*. For an original DOM such as listing 4.1, its invariant DOM can be described using the XPath expressions shown in listing 4.2.

Listing 4.1: Original DOM of a simple web application.

```
<html>
    <head>
        <title>First Test DOM</title>
    </head>
    <body>
        <h1>First Test DOM</h1>
        <p>This is a very simple DOM example that contains:</p>
        <ul id="elementlist">
            <li>A title</li>
            <li>A heading</li>
```

```
            <li>A paragraph</li>
            <li>A list</li>
            <li>A link</li>
        </ul>
        <a href="secondstate.html">Second state</a>
    </body>
</html>
```

Listing 4.2: Invariant DOM for listing 4.1.

```
//HTML
    //HEAD
        //TITLE
    //BODY
        //H1
        //P
        //UL[@id="elementlist"]
            //LI
        //A[@href="secondstate.html"]
```

4.2 Invariant Matching Algorithm

To make the algorithm work for multiple states, we use an algorithm that checks all the invariants of the previous state on the current state's DOM. This checking algorithm consists of two or three steps, which are described in the following subsections.

4.2.1 Exact Matching

Our algorithm first checks whether elements of the invariant DOM can be found by testing the XPath expression on the current DOM. These expressions search for the same element type, attribute and attribute values as the original invariant DOM contained. For example, the algorithm tries to resolve `//DIV[@id="head" class="round"]` to an element.

4.2.2 Fuzzy Matching

When the exact match fails, a fuzzy matching algorithm is used to try and find an element with roughly the same values. For example, the algorithm can match a `DIV` with `id` "content-container" to a `DIV` with `id` "contentContainer". We do this, because we sometimes want to use the invariants to match templates (simple DOM skeletons that are used as a base for building websites) against actual implementations of websites. Between a template and an actual implementation, there might be some minor differences in attribute values and we do not want to fail on these minor differences.

The algorithm works as follows. First we find elements of the same type, by searching for the XPath expression without attributes, so for `//DIV[@id="head" class="round"]` we search for only `//DIV`. This expression will most likely return a number of elements. For each of these elements, we compare the attributes and their values with the attributes and values of the invariant DOM element. This comparison is done using our own fuzzy matching algorithm.

The fuzzy matching algorithm finds the number of equal characters, found in the same order, that are contained in both inputs (the attribute value of the invariant DOM and the attribute value of the control DOM). After finding the number of equal characters, we use the Sørensen similarity index [21] to compare the similarity of these two samples.

The Sørensen similarity index is calculated using the formula $\frac{2a}{(b+c)}$ in which a is the number of elements found in both attribute values, while b and c represent the number of total elements in the invariant value and the found value respectively.

$$total\_sum = (\sum_{i=0}^{n} sørensenIndex(attribute\_value_i)) + n$$

Using the previous formula we calculate the $total\_sum$ of all Sørensen indexes of the attribute values and we also add the total number of attributes (n). If $\frac{total\_sum}{n}$, is above $2 * threshold$, we consider the elements to be equal. Note that we found this threshold by trial and error. In most cases this threshold gave the results we were looking for.

In table 4.1 we show some actual indexes. For example, if we find
`<div class="content-Container">`
using our algorithm it can match with
`<div class="contentContainer">`
This is because the Sørensen index (0.9697) summed with the total number of attributes (1) is 1.9697, so if we choose $2 * threshold$ bigger than that value, the algorithm will match these two. This can be achieved by setting $threshold = 0.85$ for instance.

Table 4.1: Fuzzy matching examples.

First Value	Second Value	Equal Characters	Sørensen Index
content-Container	contentContainer	16	0.9697
head	header	4	0.8000
mainMenu	menu	4	0.6667

4.2.3 Matching Based on Children

When at least one of the previous methods (exact or fuzzy) finds an element, we use the children of the invariant to check whether the element is really the same element in the current DOM as the element in the invariant DOM. If we can find all of these children (using the same algorithm), we check whether their parent is the element we are looking for. If that is true, we have a match: the node has the same children as in the invariant DOM, so it will probably be the same node.

Consider the invariant DOM shown in listing 4.3 as the current invariant DOM. Running a simple "exact match" against the DOM of listing 4.4 will find the `UL` element for instance. Next, the children matching algorithm is used to check whether the `UL` element has the same children as in the invariant DOM. This turns out to be true, so the `UL` element is considered to be matched.

Listing 4.3: Invariant DOM containing a menu.

```
//HTML
    //HEAD
```

```
        //TITLE
    //BODY
        //UL[@class="active"]
            //LI[@class="menuitem"]
```

Listing 4.4: Current DOM containing a menu.

```
<html>
    <head>
        <title>Menu Test</title>
    </head>
    <body>
        <ul class="active">
            <li class="menuitem">One</li>
            <li class="menuitem">Two</li>
        </ul>
    </body>
</html>
```

4.3 Deriving Invariants over Multiple States

To find DOM invariants that hold across multiple DOM states, we use a brute force algorithm, which first considers every possible invariant to be true, and when a violation occurs in a subsequent DOM state, the invariant DOM is adapted accordingly by removing the failing expression. This means the invariant DOM will shrink or stay equal in size for every iteration.

The algorithm starts with the DOM of the first page (i.e. *index*). We use this DOM to derive invariants for all its elements, resulting in our first invariant DOM. The rest of the web application is then crawled and for each new state that is encountered, the corresponding DOM tree (called *control DOM*) is used to check which elements are changed (or missing) and need to be removed from the invariant DOM. The invariants are checked using the matching algorithm described in section 4.2.

For example, consider the invariant DOM of listing 4.6. When the second state, shown in listing 4.5, is encountered, it is tested with the invariant DOM. The DOM of the second state does not contain an `A`, so it is removed from the invariant DOM, resulting in the new invariant DOM shown in listing 4.6.

Listing 4.5: DOM of the second state.

```
<html>
    <head>
        <title>Second Test DOM</title>
    </head>
    <body>
        <h1>Second Test DOM</h1>
        <p>This is the second state containing:</p>
        <ul id="elementlist">
            <li>A title</li>
            <li>A heading</li>
            <li>A paragraph</li>
            <li>A list</li>
        </ul>
    </body>
```

```
</html>
```

Listing 4.6: Invariant DOM for listing 4.5.

```
//HTML
    //HEAD
        //TITLE
    //BODY
        //H1
        //P
        //UL[@id="elementlist"]
            //LI
```

4.4 Deriving Invariants per State over Multiple Execution Runs

We can improve the quality of our invariants by deriving invariants per state. The quality of these invariants is better, because the invariants are a lot more specific. Imagine a website that has a menu and in some states it has a sub-menu as well. Using our previous algorithm, we would only be able to derive an invariant that checks whether the menu exists. Using a different invariant DOM for every state (*per-state invariants*), we can also check for the sub-menu in certain states. How we derive the state specific invariants can best be explained using figure 4.1.

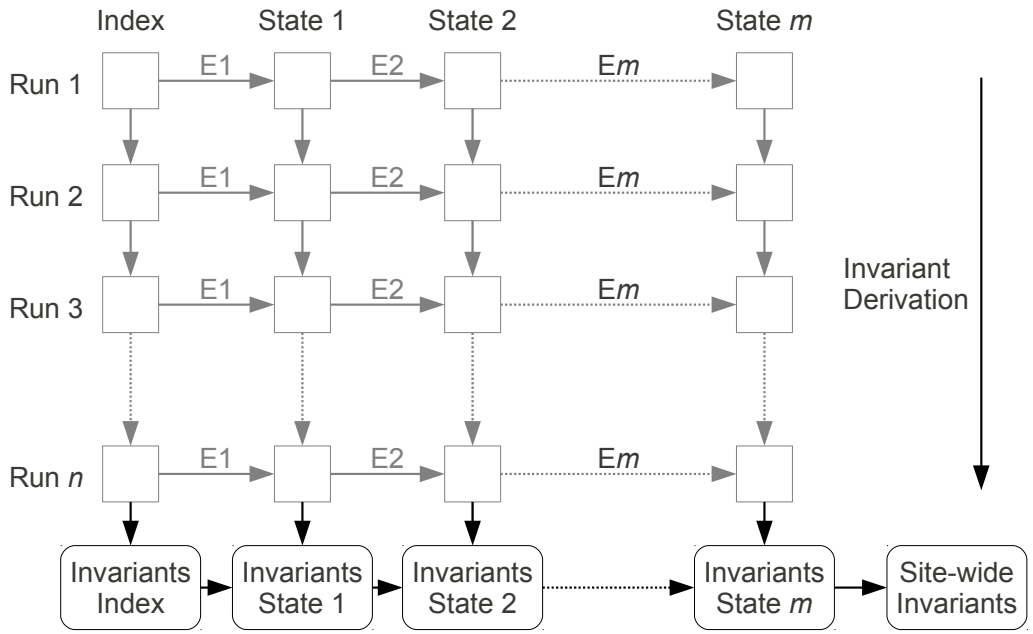

Figure 4.1: DOM Invariants per state.

Our algorithm uses a number of *runs* over the web application. These runs are basic crawls and are shown horizontally in figure 4.1. For example, firing a click event `E1` on

Algorithm 2 Finding DOM Invariants.

```
{derive a state name using a hash of the click-path}
statename ← hash(clickpath)
{can we re-use the invariants of a previous run?}
if file_exists(statename) then
    {load the invariants from a file}
    invariants ← get_file_contents(statename)
else
    {use the XPaths of all elements as invariants}
    invariants ← dom.getAllXPaths()
end if
{test the invariants, removing any failing invariants}
invariants ← invariants.testAndRemoveFailures(dom)
{save the invariants that are left to a file}
invariants.saveToFile(statename)
```

Algorithm 3 DOM Matching Algorithm.

```
match ← false
{can we find the element using an XPath search?}
foundElement ← dom.exactFind(invariant)
{if not, try to find it using fuzzy matching}
if !foundElement then
    foundElement ← domd.fuzzyFind(invariant, threshold)
end if
{if found using either fuzzy or exact matching, check children}
if foundElement then
    children ← foundElement.getChildren()
    if invariant.getChildren().equals(children) then
        match ← true
    end if
end if
return match
```

an element in the index state, will result in a state change to state 1. In each state, we run the algorithm shown in algorithm 2. This algorithm derives the invariants for that state and stores them in a file. When the next run is executed, the invariants of the previous run are loaded from the file, are checked on the (possibly modified) state using algorithm 3 and saved again.

Eventually, only the unchanging elements of the state remain in the invariant DOMs. After a number of runs, we can also derive an invariant DOM over the invariant DOMs of all states, resulting in a *site-wide invariant DOM*. This site-wide invariant DOM can be used to test states that were not found during invariant derivation. For these states, there are no state specific invariants known, so the site-wide invariant DOM is a safe fall back that still does some checking instead of nothing at all.

4.5 Testing Invariants

The invariant DOM that was found can be used to test the web application. In section 4.2, we derived an algorithm that can be used to check the invariants on a DOM. The invariant testing is done using this same algorithm. This means that invariants will fail if the children cannot be found or the fuzzy/exact match fails.

For the testing algorithm, we added another check: the ordering of the invariants is tested as well. This means that, if we find an element in the DOM, the algorithm will check whether the elements that precede it in the invariant DOM also precede the element in the current DOM.

The failures found by the testing algorithm can be saved in a report with detailed error data. For example, the current DOM, the XPath and the XPath of the children can be stored with the failure. Also, it is possible to save the current DOM so it can be compared with the invariant DOM. All of this is useful information for the programmer because it makes the errors traceable.

Chapter 5

Technical Implementation

5.1 JavaScript Invariants

We will elaborate on the technical implementation of our JavaScript instrumenter and the invariant tester in this chapter. The instrumenter and the tester consist of a Crawljax plugin and a proxy plugin.

5.1.1 Adding Instrumentation Code to JavaScript

Modifying the source code of a certain programming language can be done using a few different techniques. One of these technique is shown in figure 5.1.

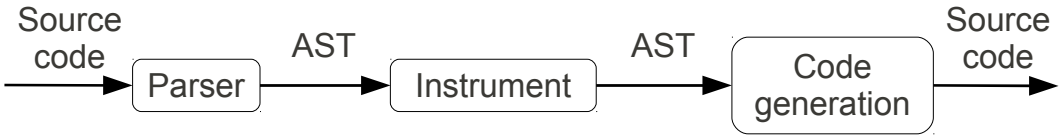

Figure 5.1: Source code modification using an abstract syntax tree.

This method uses a parser to parse the source code that needs to be modified and generates an Abstract Syntax Tree (AST) of it. The AST is a tree-like structure that can easily be traversed. The next step is adding, removing or modifying nodes in the AST. This means searching the whole AST for nodes that need modifications. When the AST modification is done, the AST has to be converted back to normal source code again.

For our specific implementation we used the parser of the open-source Mozilla Rhino JavaScript implementation.[1] Mozilla Rhino is a JavaScript implementation that is written in Java. It is a complete runtime that allows you to use JavaScript from within Java applications. Most of Rhino's features are not used by our implementation, we only needed a reliable parser that was also able to convert an AST back to its source code.

[1] `http://www.mozilla.org/rhino/`

During the development of the JavaScript instrumentation tool we found a few bugs in the implementation of Rhino. Previous versions of Rhino used a different AST format. After the last release the developers of Rhino decided they had to rewrite the AST in order to make it easier to reuse for different applications. For us, this was a great decision, because with the previous AST it was not possible to convert a modified AST back to source code while the new AST format does support this. However, the new implementation still had some small bugs. Most of them were already reported and had patches available. The others were easy to fix by ourselves and our fixes were sent to the upstream developers.

The AST generated by Rhinos parser has useful features for traversing it. We use these traversing features to search for program points where we need to add instrumentation code. Examples of these program points include:

- Function entrance.
- Function exit.
- Lines where DOM modifications are done.

The function entrance point is basically the first line of a function. Function exit can be a number of program points. It might be the last line of a function or it might be a return statements somewhere in the function body. Therefore, we search the complete function body for return statements and prepend them with the instrumentation code. The DOM modifications can be found using the patterns we described earlier in section 3.1.3. When we find a pattern like that, we add instrumentation code to the line before and the line after the pattern. Let us take a look at the example in listing 5.1. The example shows a function `calculate`, which calculates the absolute difference between two numbers.

Listing 5.1: JavaScript example.

```
var num1 = 100;
var num2 = 14;

function calculate(one, two) {
    if (one > two) {
        return one - two;
    }
    return two - one;
}

var result = calculate(num1, num2);
/* set the html value of the result div to contain the result of the calculation */
jQuery('#result').html(result);
```

The instrumented version of listing 5.1 is shown in listing 5.2. Note that the `send` and `addvariable` functions have not been defined yet. These functions are automatically added to all JavaScript files by our proxy plugin. We will explain what they do after showing its implementation in listing 5.3.

Listing 5.2: Instrumented JavaScript example.

```
var num1 = 100;
var num2 = 14;
```

```
function calculate(one, two) {
    send(new
            Array('test2.js:calculate:::ENTER',
                    addVariable('num1', num1),
                    addVariable('result', result),
                    addVariable('num2', num2),
                    addVariable('one', one),
                    addVariable('two', two)
            )
    );

    if (one > two) {
        send(new
                Array('test2.js:calculate:::EXIT',
                        addVariable('num1', num1),
                        addVariable('result', result),
                        addVariable('num2', num2),
                        addVariable('one', one),
                        addVariable('two', two)
                )
        );

        return one - two;
    }

    send(new
            Array('test2.js:calculate:::EXIT',
                    addVariable('num1', num1),
                    addVariable('result', result),
                    addVariable('num2', num2),
                    addVariable('one', one),
                    addVariable('two', two)
            )
    );

    return two - one;
}

var result = calculate(num1, num2);

/* set the html value of the result div to contain the result of the calculation */
send(new Array('test2.js:::POINT12', addvariable("jQuery('#result').html",
        jQuery('#result').html()));
jQuery('#result').html(result);
send(new Array('test2.js:::POINT13', addvariable("jQuery('#result').html",
        jQuery('#result').html()));
```

We prepend each JavaScript file with the source code that is shown in listing 5.3 and at each interesting program point we call the `send` function with an array as argument. This array contains all variables in scope. The `send` function will buffer the logs and send it to the proxy server every time its size is equal to `MAXBUFFERSIZE`. When the crawling is finished, our plugin will use Crawljax to execute some JavaScript code that will flush this buffer, because we would otherwise lose the last part of the execution trace data.

We use the `send` function to send the execution trace logs to the proxy because we had scalability issues. First we implemented the trace storing through a method that saved all traces in JavaScript arrays, but this solution could not cope with large amounts of data. By sending small parts of the trace to the proxy every few milliseconds, we do not have to store

big amounts of data anymore.

Listing 5.3: JavaScript prepended to each JavaScript file.

```
window.xhr = new XMLHttpRequest();
window.buffer = new Array();

function send(value) {
    window.buffer.push(value);
    if(window.buffer.length == MAXBUFFERSIZE) {
        window.xhr.open('POST', document.location.href
                + '?thisisanexecutiontracingcall', false);
        window.xhr.send(JSON.stringify(window.buffer));
        window.buffer = new Array();
    }
}

function addVariable(name, value) {
    if(typeof(value) == 'object') {
        if(value instanceof Array) {
            if(value.length > 0) {
                return new Array(name, typeof(value[0]) + '_array', value);
            } else {
                return new Array(name, 'object_array', value);
            }
        }
    } else if(typeof(value) != 'undefined' && typeof(value) != 'function') {
        return new Array(name, typeof(value), value);
    }

    return new Array(name, typeof(value), 'undefined');
}
```

The `send` function is called at each program point as shown in listing 5.2. The argument is a new array which has a program point identifier as the first element, followed by a number of arrays for all variables. These variable arrays contain the name, type and value of the variables.

The `addvariable` function that is shown in listing 5.3 is used to create the arrays that are send by the `send` function. They contain the name, value and type of the variables. We find out the types of the variables at runtime, because JavaScript is loosely typed. This means we could not derive the variable types syntactically in an earlier state.

5.1.2 Intercepting JavaScript Source Code

The JavaScript instrumentation code should be added to the JavaScript source files before the browser loads them. This means we need to use an intermediate "server" between the web server and the web browser: an HTTP proxy. When the browser requests a page, the request is sent to the proxy server. The proxy server can then modify the request if needed and send it to the actual web server. The response of the web server is also relayed via the proxy server. This gives the proxy server the ability to modify the response, which is precisely what we need to do.

When the HTTP proxy receives a JavaScript file, or an HTML file that contains JavaScript, as a response from the web server, it parses the file using the Rhino parser and generates an AST. The AST is modified as described in the previous section and is then converted back

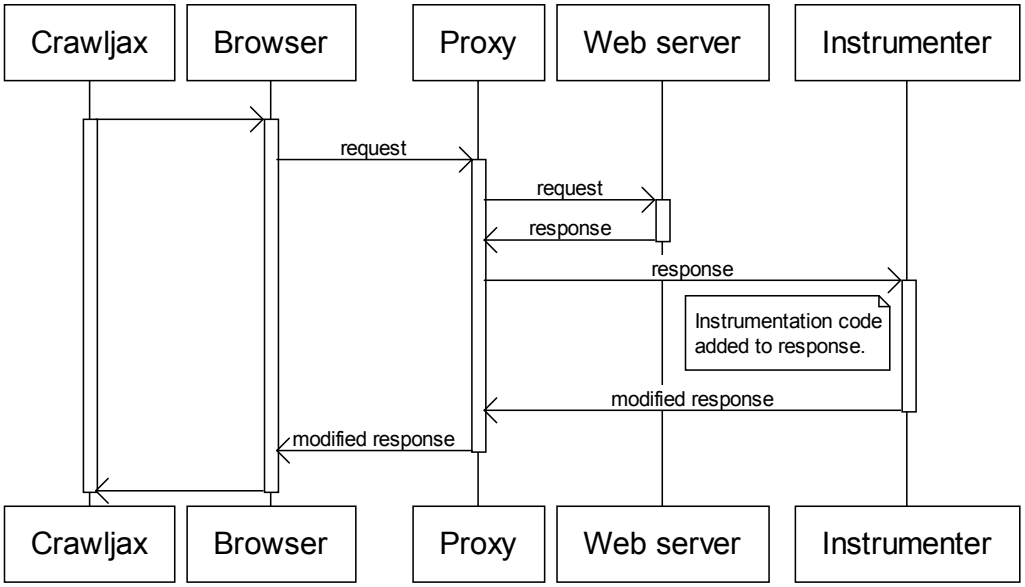

Figure 5.2: Sequence diagram of the instrumentation workflow.

to source code. The new source code is set as response and the response is sent to the web browser, as shown in figure 5.2.

Luckily for us, a proxy server based on WebScarab[2] is already integrated in Crawljax [3]. WebScarab supports plugins that modify the content of requests and responses. We therefore implemented a plugin that adds the instrumentation code to responses that appear to be JavaScript.

We encountered some problems during the development of the proxy plugins. These problems had to do with web servers not following protocol specifications precisely or wrong file encodings. For example, we saw a lot of JavaScript that was served with a different Content-Type than the official one: "application/x-javascript", so it was difficult to detect all passing JavaScript. We resolved this by just looking for the word JavaScript in the Content-Type header, but sometimes this would give problems when web servers sent JSON responses with the wrong Content-Type header.

5.1.3 Generating and Storing the Execution Trace

The proxy and the instrumentation plugin are started by Crawljax before trying to crawl a website. Everything is ready to create an execution log when the browser is finally opened. Crawljax can then start its normal crawling process. Crawljax is finished with the crawling process when all reachable states of the web applications have been visited.

After the Crawling process, our Crawljax plugin will parse all log requests received by the proxy and save the data to a so-called Daikon execution trace file. A Daikon execution

[2] `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`

trace file is a special file format used by the Daikon Invariant Detector. A very simple example is shown in listing 5.4.

Listing 5.4: Daikon execution trace file example.

```
1  decl-version 2.0
2
3  ppt allowUserAccess:::enter
4  ppt-type enter
5      variable ageOfThePerson
6          var-kind field ageOfThePerson
7          dec-type int
8          rep-type int
9
10 allowUserAccess:::enter
11 ageOfThePerson
12 11
13 1
14
15 allowUserAccess:::enter
16 ageOfThePerson
17 12
18 1
19
20 allowUserAccess:::enter
21 ageOfThePerson
22 90
23 1
```

The first line is just a version declaration for the file format. The next few lines (3-8) are a so-called declaration. A declaration specifies a program point with all the variables in it's scope. In this example, we first give this program point a name: `allowUserAccess:::enter`. The following line declares this program point as a function entry point. The lines 5-8 specify a variable with the name `ageOfThePerson` with an `int` declared type and the same represented type.

The declarations are followed by data trace records (lines 10-13, 15-18, 20-23). Each data trace record specifies all the values at a program point that was previously declared in the declarations section. This specific example has one program point and found 3 values for the `ageOfThePerson` variable at this program point. The first line of a data trace record specifies the program point. It is followed by three lines per variable which specify the name, value and modified flag respectively. Daikon allows the modified flag to always be 1 instead of sometimes 1 (for modified) and sometimes 0 (for not modified), so our implementation always sets it to 1.

Storing the trace files on disk is done by our Crawljax plugin whenever Crawljax is about to leave the current state, because the JavaScript engine is restarted when a new page loads. Not doing so would result in loss of the values encountered so far (note that they were buffered in a JavaScript array).

5.1.4 Deriving Invariants

We can run Daikon to find invariants after obtaining the execution trace. This is done by the same Crawljax plugin that saved the execution trace at an earlier stage, but it is executed at

the end of the crawling process. The output generated by Daikon is saved to a file so that we can use these invariants for testing.

5.1.5 Invariant Testing

As described in section 3.2, we decided to insert the invariant testing code using the proxy. This method is based on the method we use to instrument the code, so we could re-use a lot of that source code. In figure 5.3 we demonstrate the global workflow of this method.

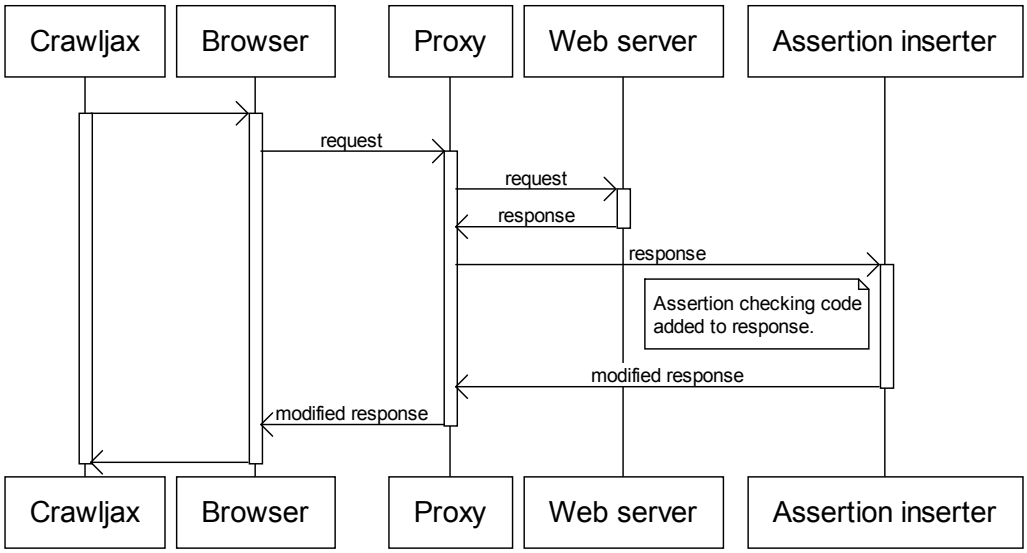

Figure 5.3: Sequence diagram of the assertion insertion workflow.

5.1.6 Handling Assertion Failures

If an assertion fails, it should be reported back to the user. The JavaScript assertion code inserts the failures in a JavaScript array. This array then contains the expression that failed, the file that contained the JavaScript code and the original line number. We developed a simple Crawljax plugin that reads out these assertion failures and adds them to the crawl report. This allows the programmer to see which assertions failed in a user friendly way.

5.1.7 Summary

We created four different plugins divided over two packages:

- com.crawljax.plugin.aji.executiontracer
- com.crawljax.plugin.aji.assertionchecker

Both packages contain an AST node visitor that is used to modify the AST of a JavaScript files. This AST is generated by a proxy plugin in order to modify the JavaScript source code before it reaches the web browser. Furthermore, both packages also contain a Crawljax plugin to read out the results of the JavaScript code that was run.

5.2 DOM Invariants

In this section we will describe our implementation of the automatic DOM invariant finder and tester we developed.

5.2.1 Invariant Finding

The algorithm we use to find DOM invariants has been described in Chapter 4, so we will only describe the technical details of its implementation in this section.

Although our algorithm only needs DOMs as input, meaning we can use a number of tools to get these, we wrote our DOM invariant finder (and tester) as plugins to Crawljax. This was done because Crawljax currently seems to be the only tool that can extract DOMs of dynamic AJAX web applications as well. "Legacy" crawlers will only download the HTML DOM from the web server, while Crawljax crawls using a web browser, thus the DOM that Crawljax hands over to its plugins might be extended or modified using JavaScript that was run by the browser.

Our plugin is executed every time Crawljax visits a new state. The plugin then gets the DOM from the browser. In every state, the plugin can access the current *click-path*. The click-path is a concatenation of the names and XPaths of all elements that were clicked to reach that state. Based on the click-path, we can derive a unique filename (by hashing the click-path) that is used to store the invariants for that state. We apply the finding algorithm to the DOM of the current state and store the results using the unique filename. In the next run, we check whether an invariant file exists for this state and use these invariants to test the current DOM.

Our plugin is executed every time Crawljax visits a new state. The plugin then gets the DOM from the browser and executes the invariant finding algorithm. The results are stored in memory until the crawling is finished. The plugin stores the invariant DOM in a file using the same format that was shown in listing 4.2, when the crawling is finished.

5.2.2 Invariant Testing

The invariant tester is also implemented as a Crawljax plugin. In every new state detected by Crawljax, the plugin loads the invariants for that state from a file. Next, the algorithm to check the invariants described in section 4.2 is executed. Failures that are found are stored in a failure report that can later be used for debugging the problems. The failures include detailed information about the failing XPath expression, the current DOM and the children of the failing XPath expression.

5.2.3 Invariant Testing at Tam Tam

For Tam Tam, the company where we developed the invariant finders and testers, we also developed a simple user interface, which is shown in figure 5.4. This user interface allows people without much knowledge about Crawljax or Java programming to test whether their HTML files comply with the company's "template". This template is basically an invariant DOM in HTML format.

Figure 5.4: DOM invariant testing interface.

Tam Tam plans to use this tool to automatically test for accessibility and Search Engine Optimization (SEO) problems. For example, for accessibility it is important that the menu is placed after the content instead of before the content. Also, because the ordering of the elements is important in these cases, our algorithm was extended with the ability to check element ordering as well.

Checking for accessibility and SEO problems is currently done manually, so our tool will help to increase productivity.

Chapter 6

Evaluation

To evaluate the correctness and effectiveness of our implementation, we did a number of case studies [31] for both the JavaScript invariants and the DOM invariants. Using these case studies we try to answer the following research questions.

1. Is it possible to automatically derive invariants over web applications?
2. What is the quality and usefulness of the invariants that are found?
3. How much manual labour is required to find invariant in an automated fashion?

6.1 JavaScript Invariants

To evaluate our JavaScript invariant derivation implementation, we conducted two case studies. In the following section we will discuss them extensively and we will eventually use the results to answer the research questions.

6.1.1 Study 1: Same Game

Our first test subject is an HTML and JavaScript implementation of *Same Game*[1]. Same Game is a simple puzzle game that is played on a board which is randomly filled with blocks of different colours. Groups of neighbouring blocks of the same colour can be removed by clicking on them. Blocks that are not supported by other blocks anymore will fall down and if empty columns appear they will be replaced with the columns right of it.

The implementation of this game was created by two students of our group and one of the goals was to write clear and concise code. The implementation was created using the jQuery[2] JavaScript library, consists of about 250 lines of JavaScript and can be found at `http://crawljax.com/same-game`. A screenshot is shown in figure 6.1.

[1] `http://en.wikipedia.org/wiki/SameGame`

[2] `http://jquery.com/`

Table 6.1: Found JavaScript invariants.

Web Application	Same Game	Tunnel Game
Total Lines of Code (without jQuery)	~ 250	~ 370
Automatically Found Invariants		
Total Number of Invariants	150	3850
- Function Entry	53	1531
- Function Exit	91	2319
- DOM Manipulations	6	0 *
Unique Invariants	34	291
Manually Found Invariants		
Total Number of Invariants	30	20
- Function Entry	13	10
- Function Exit	7	3
- In Middle of Function	10	7

** Due to a bug in our implementation we could not find DOM manipulation invariants for Tunnel Game.*

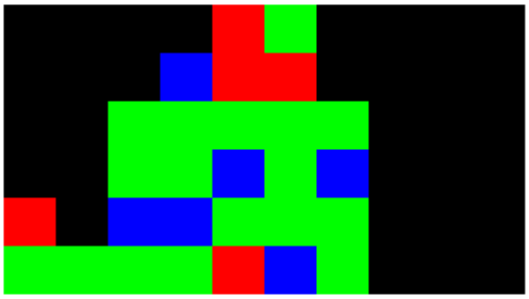

Figure 6.1: Test case 1: Same Game.

Setup

We asked the two developers of the game to examine their game thoroughly and document possible invariants. We converted these invariants to actual JavaScript assertions and we inserted these in the source code. In total, this cost about 70 minutes of effort. This new version of the game can be used to introduce a bug in and test whether the manually found invariants are able to detect it.

Next, we used our tool to derive invariants automatically. To do this, we had to write a runner class. A runner class is a simple class with a main method that is used to configure Crawljax through its API and load all needed plugins. Furthermore, we had to write a very simple Crawljax plugin that modifies the candidate clickables list to contain only one random element in each state. This was done to ensure Crawljax would not crawl endlessly, because the game does not support proper backtracking. That is, when the last state is reached, it is not possible to backtrack to the previous state by refreshing the page and clicking the same element, because the board is randomly generated when the page is loaded.

Writing the plugin that reduced the number of clickables to one, took about one minute of manual labour. Writing the runner class with the correct configuration options such as which elements to click took about three minutes of manual labour. We did quickly found out which elements could be clicked, because they had a CSS class "clickable" in the DOM tree. This means that finding invariants for this web application took about four minutes, while developing the application took about four hours. So, the manual labour involved to find the invariants is less than two percent of the development time.

Automatically Deriving Invariants

The runner class described in the previous section was used to find invariants. We let Crawljax crawl approximately 50 states, where it found 49 clickable elements (294 examined elements). Eventually, the execution trace became 11 MB in size.

The results are shown in table 6.1. It shows the total number of invariants found automatically and manually. Furthermore, these are divided in three subtypes, namely function entry, function exit and DOM manipulations for automatically found invariants or "in middle of function" for manually found invariants.

In total 44 program points were inspected, of which 10 where function entry points, 18 were function exit points and 16 were DOM modification points. For these program points, a total of 150 invariants, 34 unique, were found. For the function entry and exit points, we found approximately 5 assertions per point.

A number of useful invariants were found. One of them is an invariant that considers the `height` and `width` variables that define the board size to be equal to some constant value. It was interesting to see this, because this was an invariant the developers came up with themselves as well.

Another interesting invariant that was found was for the method that marks a cell as "to be removed". This method has three parameters, the two coordinates and the colour of the cell that was originally clicked. Our tool came up with invariants that made sure the x and y coordinates were always valid, i.e. `x >= 0`, `width > x`, `y >= 0` and `height > y`. These invariants are very useful to detect off-by-one errors. Furthermore, it found an invariant to make sure the `value` (colour) argument was a valid colour: `(value == 1.0 || value == 2.0 || value == 3.0)`. This invariant also makes sure we cannot mark cells that were already empty, because those cells have a colour value of zero.

Furthermore, some interesting DOM modification invariants were detected. For example, a function that adds the clickable class to elements that, according to the game rules, should be clickable was extended with an invariant check to verify the class was actually added to the elements.

Finding Bugs

Next, we wanted to find out how useful these invariants actually were. We asked a programmer to introduce a number of bugs in the game without telling us what they were. The bugs are listed in the first column of table 6.2.

Table 6.2: Injected bugs in the Same Game.

Injected Bug	Detected Using: Automatic Invariants	Detected Using: Manual Invariants
1. The `x` and `y` arguments of a `mark` function call were swapped. This function call is executed after clicking an element and is used to mark the cell for removal.	Yes	Yes
2. Each cell had a dynamically generated HTML attribute that contained the x and y coordinates of that cell separated by a dash. This dash was removed, so the x and y coordinates were concatenated without a separator.	No	No
3. The `updateBoard` function, partly shown in figure 6.2, draws the board and checks whether the game has finished. The last if statement checks whether there are still coloured cells left on the board. This is done using `numCells[i]`, which contains the number of cells left for colour `i`, whereby zero represents the "empty colour". The check whether all colours are gone was modified to always evaluate to true by changing the non-equal check to an equality check. This change has the annoying effect that every time a cell is clicked, a "game won" message is displayed.	Yes	No
4. `equalNeighbour` considers unequal neighbours as equal. Single cells can now be clicked (and removed).	No	No
5. Negative board `width` value.	Yes	Yes
6. In `randomValues`, the colours values array was changed to have four extra colours.	Yes	Yes
7. The `clearChecked` function call was removed in the onclick handler.	No	Yes
8. In `updateBoard` initial value of the number of clickables was set to one instead of zero.	No	No
9. The length of the colours array was modified by removing one element.	Yes	No
10. In the onclick handler of the cells, the `equalNeighbour` check was negated.	No	No
Detected bugs	50%	40%

After introducing these bugs one at a time, we ran Crawljax with the assertion checker and we ran the versions of the game that had manual invariant assertions added. In table 6.2 we list the results of these runs. The first column lists the bug. The second column indicates whether or not the automatically found invariants were able to detect the bug and the last column indicates whether the manually found invariants were able to detect the bug.

Of the ten bugs that were introduced, five were detected using the automatically found invariants. Using the manual invariants, only four bugs were detected.

Looking at the amount of manual effort to find the invariants, the manual invariants found less bugs and cost more development time than the automatically found invariants.

6.1.2 Study 2: Tunnel Game

Our second test subject is an HTML and JavaScript implementation of a tunnel game[3] developed by Christian Montoya. In this game you control an airplane and the objective is to avoid hitting the wall, which moves randomly. A screenshot of the game is shown in figure 6.3. It is written using jQuery, just like Same Game, and consists of about 370 custom lines of JavaScript code.

Setup

For this game, we asked the two developers of Same Game to examine Tunnel Game and document possible invariants. Again, we converted these invariants to actual JavaScript

[3] `http://arcade.christianmontoya.com/tunnel/`

```
/* redraw board and check if game is
   finished */
function updateBoard() {
    /* counter for number of cells that
       can be clicked */
    var clickables = 0;
    /* count all colours in this array */
    var numCells = new Array();
    for(var i = 0; i < colours.length; i++) {
        numCells.push(0);
    }

    for(var y = 0; y < height; y++) {
        for(var x = 0; x < width; x++) {
            ...
            numCells[board[y][x]]++;
            /* add onclick events only if
               there is an equal coloured
               neighbour */
            if(equalNeighbour(x, y)) {
                clickables++;
                ...
            }
            ...
        }
    }

    /* check if all colours are gone */
    if(numCells[0] != (width * height)) {
        /* check if there are no clickables
           anymore */
        if(clickables == 0) {
            /* still some coloured cells left, so: GAME OVER */
            ...
        }
        return;
    }
    /* apparently, there is nothing left, so: GAME WON */
    ...
}
```

Figure 6.2: The updateBoard function.

invariants and inserted them in the game. In total this cost about 60 minutes of manual effort.

To automatically derive invariants for this game, we had to write a runner class. A problem we encountered was the fact that Crawljax does not support emulating mouse movement (because this is not available in WebDriver). This means we could not let Crawljax completely automatically play the game. However, we could still automatically gather quite some data in the following way.

The airplane of the game can be moved to the left or to the right by moving the mouse to the left or right of the screen. So when we place the mouse in the middle before starting the game, the airplane will stay in the middle of the tunnel. The random movement of the tunnel

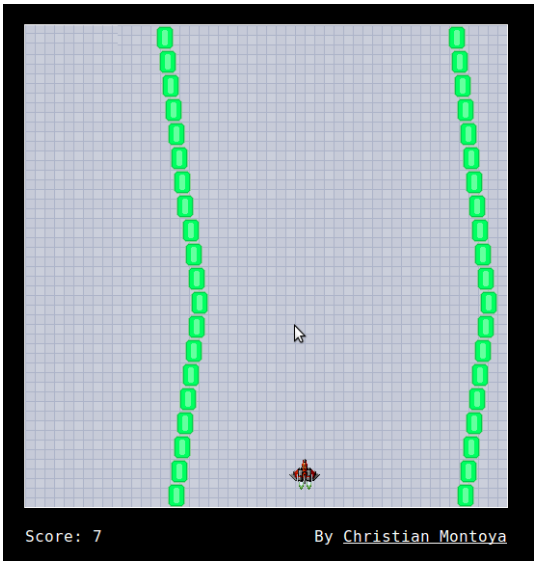

Figure 6.3: Test case 2: Tunnel Game.

is not that big, so we can get quite a good score without moving the plane. Furthermore, if we place the mouse to the left or right of the screen, the plane will crash into the wall, but this is data we need as well.

The runner class only consists of a statement to click the start button and a statement to set the default waiting time after the click. Writing this runner cost us less than a minute of manual labour.

Automatically Deriving Invariants

Using the runner class described in the previous section, we were able to gather quite some data. In total we did fifteen runs, of which two were "manual runs". These manual runs were done by starting Crawljax and then controlling the mouse to make sure we got a higher score. This means that gathering the data and writing the runner class cost us about seven minutes of manual labour.

Our execution trace eventually grew to 63 MB of data. The execution trace was used as input for Daikon and the results are shown in table 6.1. In total 102 program points were inspected, 51 entry points and 51 exit points. The total number of invariants found was 3850, of which 291 were unique. Approximately 37 assertions were found per program point.

Some interesting invariants were found, for example, Daikon derived checks to verify the plane was always positioned between the two walls and the space between the two walls was always big enough (either 280, 300 or 320 pixels). However, some false positives were also found. One good example is the check `score < ship_x`. When the user plays long enough, he might get a score that is bigger than the x coordinates of the ship, which will result in failures. To avoid these kind of false positives, a more extensive execution trace

will help, because it will invalidate these invariants.

Finding Bugs

To evaluate the quality of the found invariants, we asked a programmer to introduce a number of bugs. These bugs were not explained to us, so we had to use the invariants to find them. The bugs are listed in the first column of table 6.3.

Table 6.3: Injected bugs in Tunnel Game.

	Detected Using	
Injected Bug	Automatic Invariants	Manual Invariants
1. In the `updateTunnel` method, we changed the calculation that was used for random wall movement.	Yes	Yes
2. The score counter was changed to decrement instead of increment.	Yes	No
3. A variable that is used for the background movement was change dfrom zero to minus one.	Yes	Yes
4. The code that kills the player when he hits a wall was removed.	No	Yes
5. The starting score of the game was changed to be negative instead of zero.	Yes	Yes
6. Code that is used to verify that the tunnel is never wider than a certain value was removed. This means you can keep the airplane in the middle of the screen without ever touching the tunnel.	No	No
7. The code that modifies the position of the plane was removed. This means the plane cannot be moved anymore.	Yes	No
8. The code that moves the wall blocks was removed. Only the first two blocks still moved (they cannot hit the player).	Yes	No
9. The code that makes sure the right_wall moves the same as the left_wall was removed.	Yes	No
10. The score increment rate was changed to be much faster, almost equal to the frame rate.	Yes	No
Detected bugs	80%	40%

Again we introduced these bugs one by one. We ran Crawljax and the manual invariant versions over the modified game and recorded all results. They can be seen in table 6.3. The manual invariants detected four out of ten bugs and the automatically found invariants detected eight out of ten bugs. However, it should be noted that bug number ten was only detected because a very high score was reached, much higher than the scores that were reached during invariant derivation. Had we created a more extensive execution trace, we would not have had any invariant failures for that last bug.

6.1.3 Results

The results of the previous two case studies seem very positive. Using these results, we come to the following answers for our research questions about JavaScript invariant derivation and testing.

1. It is possible to automatically derive invariants on JavaScript variables in web applications.

2. The found invariants are of such quality that it is possible to use them for automatic bug detection.

3. The amount of manual labour required to automatically derive invariants is much less than the development time of the applications. For these two case studies, it is most likely less than two percent of the total development time. Also, compared to the manually found invariants, the time to find the invariants is much less, and the quality is better as well.

6.2 DOM Invariants

To evaluate our DOM invariant derivation implementation, we conducted three case studies. In the following section we will discuss them extensively and we will eventually use the results to answer the research questions. For all studies we choose the $threshold$ value of the fuzzy matching algorithm (explained in section 4.2.2) to be 0.85.

Table 6.4: DOM evaluation results after 3 runs.

Web Application	The Organizer	Bookstore	Yellow Pages
Number of States	20	67	93
Total Number of Elements	2957	8914	10731
Total Number of Invariants	2389	4717	4843
Minimal Number of Invariants	118	68	41
Maximal Number of Invariants	120	110	57
Number of Injected Bugs	4	4	4
Number of Detected Bugs	4	4	3

6.2.1 Study 1: The Organizer

The first case study we did was conducted on a simple application called The Organizer.[4] The Organizer can be used as a task manager and organizer. For example, you can add to-do items, create appointments and add contacts. The application is built as an exercise in a course book and is therefore available as an open source project. The Organizer was written in Java and a screenshot of the index page is shown in figure 6.4.

Setup

Writing the runner class for The Organizer was pretty simple. We used the "clickDefault-Elements" functionality, which covers most clickables in standard web applications. In total, writing the runner class took less than two minutes of manual labour.

Deriving Invariants

Using the runner class, we did three runs to derive the per-state invariants. We limited the number of states to twenty. The resulting numbers are shown in table 6.4. These results are based on three separate runs. The first row indicates the number of states that were crawled

[4] `http://www.apress.com/book/downloadfile/2931`

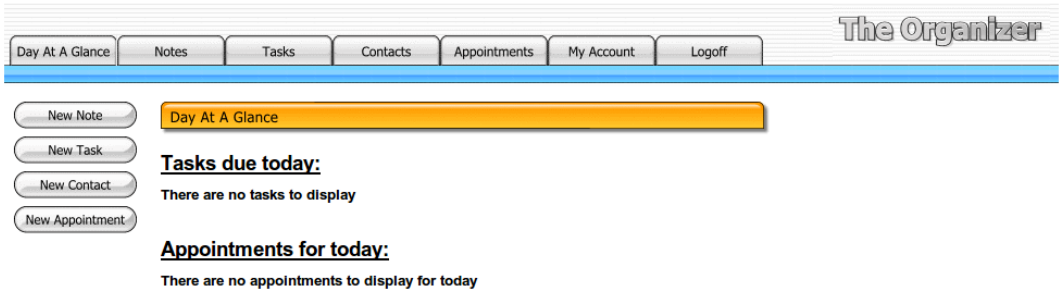

Figure 6.4: Test case 1: The Organizer.

in every run. Next, the total number of elements found in every run is reported. In the third row, we show the total number of invariants found for the complete application. The next two rows report that every state has at least the minimal number indicated and at most the maximal number of invariants indicated.

For The Organizer, all states had almost the same number of invariants, only varying two invariants at most. This might be because the DOM is not actually modified by The Organizer, but elements are just hidden using CSS.

Finding Bugs

To test the quality of the found invariants, we introduced a number of bugs in the DOM. These bugs were not added all at once, but one at a time. Meaning we introduced a bug, ran an invariant test, fixed the bug, introduced another bug, and so on. We introduced the following bugs.

1. We removed the menu that is shown on all pages.
2. We removed the image tag of the logo from the header.
3. We moved the menu, which is located in a table row, to the bottom of the table. This means that the menu is shown at the bottom of the page instead of at the top.
4. We re-ordered elements of the menu.

For each bug, we had at least one invariant failure. This means that each of the bugs was detected correctly. Also, the failures included enough details to easily pinpoint the exact location of the missing element or the position where the element should be.

6.2.2 Study 2: Bookstore

The second study was conducted on a web application called Bookstore.[5] Bookstore is an open source web application that can be used to sell books online. It includes a user registration system, product voting, categories, shopping cart and administration of various web shop aspects. A screenshot of Bookstore is shown in figure 6.5.

Figure 6.5: Test case 2: Bookstore.

Setup

For this test case, we re-used the runner we developed for The Organizer. Therefore, the same amount of manual labour as before was needed.

Deriving Invariants

Using our runner class, we did three runs to derive the invariants. We did not limit the total number of states, so the crawls took quite some time (around six minutes). Final results are shown in table 6.4 as well. As you can see, the number of invariants per state differs a lot more than for The Organizer. In some states, only 68 invariants could be detected, and in others 110 invariants were found. The average number of elements per state was less than the average we found for The Organizer.

[5] `http://gotocode.com/apps.asp?app_id=3`

Finding Bugs

For this case, we also tested the quality of the invariants by introducing bugs. We used the same method as for The Organizer and we introduced the following bugs one by one:

1. We removed the search block that was displayed in the sidebar.
2. We moved the search block that was displayed in the sidebar to the bottom of the sidebar.
3. We removed the registration form of the registration page.
4. We re-ordered elements of the registration form, for example, we switched e-mail and last name.

Finding these bugs was successful as well. For each bug, we had at least one failure in the corresponding state.

6.2.3 Study 3: Yellow Pages

The third study we did was done on Yellow Pages,[6] another open source web application, which is shown in figure 6.6. Yellow Pages is an application in which the user can find contact information by browsing different categories or searching for specific terms.

Figure 6.6: Test case 3: Yellow Pages.

[6] http://gotocode.com/apps.asp?app_id=4

Setup

For our last case study, we also used the same runner class as before. We did not limit the number of states because crawling the application did not take very long.

Deriving Invariants

Our runner class was used to do three runs. This application had the biggest number of states (93), but because the DOMs were small, crawling took less time than the previous cases.

The results of the runs are shown in table 6.4. Variations in the number of invariants per state are small, because most states are relatively similar.

Finding Bugs

We introduced the following bugs one by one in the Yellow Pages application.

1. We removed the search button that is displayed in the search form.
2. We removed the logo that is displayed in the header.
3. We removed the item count that is shown on result pages. Normally it would say "10 items found", for example.
4. We removed enclosing `TR` and `TD` elements of a link in the menu.

The first three bugs were detected correctly, however number four was not. Number four shows a problem in our implementation. Because we use XPath expressions, these might turn out to be fairly generic. This means that, for the DOM shown in listing 6.1, it is only possible to detect whether all `LI` items are missing, because both items have the same XPath expression: `//LI[@class="someitem"]`. Testing this XPath expression on the DOM shown in listing 6.2 will not fail, because one element is still returned. The fourth bug hit this problem exactly. For one menu-item, the `TR` and `TD` elements were removed. However, there were other menu-items that still had the `TR` and `TD` elements, meaning the XPath expression did not fail.

Listing 6.1: Sample DOM to demonstrate possible XPath problem.

```
<html>
    <head>
        <title>First Test DOM</title>
    </head>
    <body>
        <ul id="elementlist">
            <li class="someitem">A title</li>
            <li class="someitem">Sometext</li>
        </ul>
    </body>
</html>
```

Listing 6.2: Second sample DOM to demonstrate possible XPath problem.

```
<html>
    <head>
        <title>First Test DOM</title>
    </head>
    <body>
        <ul id="elementlist">
            <li class="someitem">A title</li>
        </ul>
    </body>
</html>
```

6.2.4 Results

The results of our DOM invariant case studies seem very positive as well. Using these case studies, we come to the following answers for our research questions about the DOM invariant derivation and testing.

1. It is possible to automatically derive invariants over the DOM of web applications. In our case studies we found a big number of invariants for the applications.

2. The quality of the found invariants is such that it is possible to use them for automatic bug detection.

3. The amount of manual labour required to actually derive invariants is much less than the development time of the applications, typically less than 5 minutes are needed (only the time to write a runner class).

Chapter 7

Discussion

In this chapter we will discuss the results of our evaluation and our implementation in general.

7.1 Applicability

During the development of our JavaScript invariant deriver we found out it cannot find satisfying results for all types of web applications. For example, simple websites that use JavaScript to only show and hide HTML elements will probably not have any useful invariants. We think the best application are computation intensive web applications that do most of the computation on the client side. A good example of such an application is the Same Game we tested. It has code that checks the board after every click, code that modifies the board if needed etc.

Our DOM invariant deriver is applicable to all kinds of web applications, varying from static web pages to very dynamic web applications. The only applications that are probably not suitable, are applications that dynamically generate unique `id` values for all elements. This means that a new version might have a completely different `id` as the previous version. A good example is an application that includes version numbers in its `id` attributes, meaning all `id` values change when a new version is released.

7.2 Highly Dynamic Web Applications

We did not conduct an actual case study on a highly dynamic web application, because we could not find an open source one and it is difficult to introduce bugs if you cannot simply modify the source code files. However, we were able to detect invariants over the DOMs of several sites such as Twitter,[1] Slashdot[2] and Nu.[3] We did so to test whether detecting invariants in different runs would result in changes in the invariant DOMs, because in our

[1] `http://twitter.com/`
[2] `http://slashdot.org/`
[3] `http://nu.nl/`

case studies it did not. For these websites, the invariant DOM of certain states did indeed change, so the subjects of our case studies were probably not dynamic enough.

7.3 Generated JavaScript

A number of frameworks exist to automatically generate most client-side code. An example of such a framework is Google Web Toolkit (GWT).[4] While our approach seems to work fine for hand-written JavaScript code, it does not for generated JavaScript. Invariants or assertion failures found in generated code are not meaningful to developers, because they cannot easily trace where these errors originated from in their original source code (Java for GWT applications). This means you might be able to find errors, but it can be very difficult to locate where these errors come from in the original source code.

7.4 Implementation Limitations

Our testing and invariant derivation methods are fairly generic, they can be used with manual web application execution or using automation tools such as Crawljax or Selenium. Our current implementation is developed as several plugins to Crawljax, so it is limited by the abilities of Crawljax. With this implementation, we can only obtain execution traces of applications that can be crawled using Crawljax. For example, applications that need a lot of drag-and-drop events can currently not be crawled using Crawljax, because WebDriver does not allow us to programmatically produce such events. This means that we cannot obtain a good execution trace of such application using our current implementation, so we cannot derive good invariants either.

The use of a proxy to intercept JavaScript source code infers a limitation as well: it can only be used for web applications that use no encrypted connections. When using an HTTPS connection, all data is encrypted by the browser and decrypted by the web server. This means our proxy cannot identify any JavaScript that passes and thus is not able to instrument it.

Finally, during the evaluation of our tools we found out that minified, compressed or obfuscated JavaScript files might not be parsed correctly with Rhino. This means that we cannot obtain an execution trace for applications where we do not have access to the original JavaScript files.

7.5 Comparison to Existing Tools and Techniques

To the best of our knowledge, we are the first to create execution trace of JavaScript code and use these to derive invariants. Instrumenting JavaScript is not a new invention and we based our implementation on the proxy technique described by Haruka Kikuhi et al [14].

For deriving DOM invariants there exists another tool: DoDOM. Our tool differs from DoDOM in a few aspects. DoDOM cannot automatically crawl through a web application

[4] `http://code.google.com/webtoolkit/`

but needs a user to interact with the web application. DoDOM derives invariants over a number of state transitions, while our tool derives invariants per state. Finally, our tool looks at DOM elements and their children, while DoDOM seems to look at DOM elements and their content, ignoring the children.

7.6 Threats to Validity

Concerning *external validity*, our study is performed on a limited number of web applications. Generalizing the results based on only these studies might harm validity, although the selected cases represent the type of web applications targeted by our research. We did conduct more case studies on different kinds of web applications, but we had problems in correctly deriving invariants due to bugs in Rhino, Daikon and our own tool implementation. A list of these failing cases and their problems is given in appendix A. Finally, we can only detect DOM manipulations done using the jQuery library. This might harm the external validity as well.

With respect to *internal validity*, we tried to minimize the number of bugs in the tools developed by writing JUnit tests for the JavaScript invariant deriver and tester. However, we also various third party tools and libraries, such as Daikon and Rhino. We did encounter several problems in some of these, so these libraries and tools might harm the internal validity.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The introduction of our plugins to automatically derive and test invariants in web applications proved to be a useful one. The results of our case studies are promising. For the JavaScript invariants, data intensive applications seem to benefit from our invariant deriver when it comes to automatic regression testing. Furthermore, our DOM invariant deriver detects undesired DOM modifications and can therefore help improve the quality of web applications by automatically testing for regressions.

In short, the contributions of this master thesis include:

1. A method for instrumenting JavaScript code and tracing program state changes, including programmatic (through JavaScript) DOM element and attribute manipulations.

2. An approach for automatically deriving JavaScript invariants in modern web applications.

3. An automated method in which derived invariants are used for testing web applications.

4. Automatic DOM invariant derivation for individual states, over one or several runs and for all states combined (*site-wide invariants*).

5. Implementation of these methods in a number of plugins to Crawljax and the WebScarab proxy.

8.2 Future Work

Future work can be done in a number of different directions. In a practical sense, the current quality of our plugins should be improved. Some parts of the plugins are underdeveloped, for example, the current version can only detect DOM manipulation patterns done by jQuery based web applications. Furthermore, some outstanding bugs should be fixed.

Future research could include improvements to our DOM invariant deriving and testing algorithms to avoid the problem we discussed in 6.2.3. This might be done by including the number of elements an invariant XPath should return or by making the XPath expressions more specific to always resolve to one element.

Also, future work might generalize our DOM invariant ideas to include something like regular expressions. This might allow one to automatically verify the number of columns of a table is correct, while the number of rows might vary.

Finally, it might be useful to modify the DOM invariant deriving algorithm to be more flexible when an element cannot exactly be found. Simplifying the expression might then be a better solution than completely removing it. For example, when `//DIV[@id="main" @class="active"]` cannot be found using the exact matching or fuzzy matching algorithm, it is currently removed from the invariant DOM. However, it might be possible that `//DIV[@id="main"]` does match without any problems. When it does match, it means that replacing the original expression with the simplified one is a better solution.

Bibliography

[1] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMS. *Software and Systems Modeling*, 4:326–345, 2005.

[2] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WWW2002)*, 2002.

[3] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. Automated security testing of web widget interactions. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 81–91, New York, NY, USA, 2009. ACM.

[4] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[5] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 281–290. ACM, May 2008.

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[7] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[8] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-

11-02, University of Washington Department of Computer Science and Engineering, Seattle, WA, November 16, 1999. Revised March 17, 2000.

[9] Jesse J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, February 2005.

[10] Markus Geimer, Sameer S. Shende, Allen D. Malony, and Felix Wolf. A generic and configurable source-code instrumentation component. In *ICCS 2009: Proceedings of the 9th International Conference on Computational Science*, pages 696–705, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Dick Grune, C. Jacobs, Koen Langendoen, and Henri Bal. *Modern Compiler Design*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[12] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[13] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2007. ACM.

[14] Haruka Kikuchi, Dachuan Yu, Ajay Chander, and Hiroshi Inamuraand Igor Serikov. Javascript instrumentation in practice. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. A case study-based comparison of web testing techniques applied to Ajax web applications. *Int. Journal on Software Tools for Technology Transfer*, 10(6):477–492, 2008.

[17] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In D. Schwabe, F. Curbera, and P. Dantzig, editors, *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, July 2008.

[18] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers*, pages 210–220. IEEE Computer Society, 2009.

[19] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[20] Bertrand Meyer. Seven principles of software testing. *Computer*, 41(8):99–101, 2008.

[21] D. Mueller-Dombois and H. Ellenberg. *Aims and methods of vegetation ecology*. Wiley, Chichester, UK, 1974.

[22] Lasse Reichstein Nielsen. Definition of a JavaScript closure. `http://www.jibbering.com/faq/faq_notes/closures.html`, November 2009.

[23] Karthik Pattabiraman and Benjamin Zorn. DoDOM: Leveraging DOM invariants for web 2.0 application reliability. Technical Report MSR-TR-2009-176, Microsoft Research, 2009.

[24] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.

[25] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, pages 87–102. ACM, 2009.

[26] Arun Ranganathan. ECMAScript File API. `http://dev.w3.org/2006/webapi/FileAPI/`, November 2009.

[27] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11–43, 2007.

[28] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), Research Papers*, 2001.

[29] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Stephen McCamant, Feng Mao, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[30] Alan M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69, Cambridge, UK, 1949. University Mathematical Laboratory, Cambridge University.

[31] Robert K. Yin. *Case Study Research: Design and Methods, Third Edition, Applied Social Research Methods Series, Vol 5*. Sage Publications, Inc, 3rd edition, December 2002.

Appendix A

Failed Test Cases

We conducted a lot more case studies than the ones mentioned in our evaluation. However, due to bugs in Rhino, Daikon or our own tool implementation, we were not able to derive invariants correctly. This appendix lists these cases and the problems we encountered.

1. `http://dwpe.googlecode.com/svn/trunk/charting/index.html`
   This is a simple library that can convert HTML tables to charts. Daikon gives an output error when using JavaScript output format, this is a bug in my code probably.

2. http://www.bramstein.com/projects/typeset/
   Latex line-breaking algorithm implemented in JavaScript with canvas.
   Fails with a ”too much recursion” error produced by WebDriver. Also, when excluding all files, still sort-of-crashes because of the inline javascript (in the HTML page).

3. `http://www.andrew-hoyer.com/exp_src/cloth.html`
   JavaScript demonstration of a simple three dimensional drawing library.
   Mostly constants found, never got to the point of actually drawing the cloth when run with instrumentation code.

4. `http://olympisch.nl/`
   Social media website of the Olympic Games 2010.
   Daikon crashes.

5. `http://www.xs4all.nl/~peterned/3d/`
   Three dimensional demonstration.
   Apparently javascript is compressed, so only some invariants for the compressor.

6. `http://hernan.amiune.com/labs/harmonograph/animated-harmonograph.html`
   Animated Harmonograph.
   Crashing script in browser (aka ”The following script is unresponsive, do you want to stop it?”).

7. `http://github.com/mrdoob/three.js`
   3D JavaScript library.
   Cube demo crashes Java (out of memory), even with 1 GB for the Java process.
   Other demo's won't even run smooth without instrumentation code.

8. `http://people.iola.dk/olau/flot/examples/`
   Charting library.
   Out of memory crashes.

9. `http://vis.stanford.edu/protovis/ex/force.html`
   This demo doesn't even work in Firefox without instrumentation code, it runs too slow.

10. `http://labs.dextrose.com/`
    Numerous demo's, none of them seem to have enough custom JavaScript code.

11. `http://gamequery.onaluf.org/demos/4/iframe.html`
    jQuery-based plugin.
    NullPointerExceptions, because of a bug in Rhino.

12. `http://www.arcinspirations.com/kobe/`
    Problems with Daikon, stuff like "Exceptional exit" & EmptyStackExceptions.

13. `http://www.fernando.com.ar/memo/`
    Another jQuery game.
    Almost no invariants, because almost no custom JavaScript code.

14. `http://bonadiesarchitect.com/`
    Informational website.
    Almost no invariants, because almost no custom JavaScript code.

15. `http://www.alexweber.com.br/minesweeper/`
    Minesweeper game.
    Most important js files are minified/obfuscated.

16. `http://www.alexweber.com.br/memorygame/`
    Memory game.
    The most important js files are minified/obfuscated.

17. `http://www.freejavascriptgames.info/games/snake.html`
    Snake game.
    Quite some invariants, but all over arrays and all seem not useful (only for some specific executions, these invariants were true, if you do more executions they will fail).

18. `http://pleaserobme.com/`
    NullPointerExceptions, because of a bug in Rhino.

Appendix B

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Invariant An invariant is an expression defined over variables of an algorithm or software program that should evaluate to true on every function entry or exit point [19].

Assertion A basic check for a condition. If the condition fails, the program will stop executing most of the time. Assertions can be used to test invariants for validity.

XPath XML Path Language. Short expressions that can be used to query a DOM for certain elements.

HTML HyperText Markup Language. The most popular markup language that is used for creating web applications.

JavaScript JavaScript is an implementation of the ECMAScript language standard and is typically used to create interactive web applications.

DOM Document Object Model. The structural representation of an HTML page.

AJAX Asynchronous JavaScript and XML. A combination of techniques used to improve the user experience for websites by not completely refreshing the whole page.

Crawljax A tool that can be used to automatically crawl highly dynamic web applications. Crawljax can be extended by developing plugins.

Daikon A tool that can be used to derive invariants over data of a program execution (see *execution trace*).

AST Abstract Syntax Tree. A tree representation of the abstract syntactic structure of source code written in a programming language.

HTTP proxy In its simplest form, an HTTP proxy sits between a webserver and the webbrowser. It forwards requests from the webbrowser to the webserver and sends responses from the webserver to the webbrowser.

Execution Trace A file that contains variable names, types and values of a program execution.

Program Point A point in a program (basically a line in the source code) where data is saved to an execution trace.

Instrumentation Code Code that is added to a program, which can generate an execution trace.