USING PROPERTY-BASED TESTING, WEIGHTED GRAMMAR-BASED GENERATORS AND A CONSENSUS ORACLE TO TEST BROWSER RENDERING ENGINES AND TO REPRODUCE MINIMIZED VERSIONS OF EXISTING TEST CASES

by

JOEL DAVID MARTIN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2019

Supervising Committee:

Farhad Kamangar, Supervising Professor

David Levine, Supervising Professor

Manfred Huber

Gergley Zaruba

ACKNOWLEDGMENTS

Thank you to the members of my Supervising Committee for their support and guidance. In particular, I would like to thank David Levine who spent many long hours with me discussing the research and helping with the numerous large and small details that were necessary to complete my thesis. Many thanks to my employer, Viasat, for the financial investment toward my education. Thank you to my supervisors at Viasat, Ingolf Krueger and Markus Kaltenbach, for their support and encouragement throughout my graduate education. I would like to apologize for the harm done to numerous figurative trees during the completion of this research. As a thank you to the reader(s) of this dissertation, I offer you a gift. Contact me with proof that you have read this work and I will send you a prize (limited quantities available). Thank you to Aaron Brooks for being a sounding board for many of the ideas that were explored in this research. Thank you to Matt Oquist, Ingolf Krueger, and Rebecca Martin for their detailed reviews and the valuable feedback that they provided.

DEDICATION

I dedicate this work to my brilliant and beautiful wife, Dr. Rebecca Martin. She worked hard to ensure that I would be successful in this endeavor. Let it be noted, for posterity, that she received her doctorate 8.5 years prior to me receiving mine.

ABSTRACT

Using Property-Based Testing, Weighted Grammar-Based Generators, and a Consensus Oracle to Test Browser Rendering Engines and to Reproduce Minimized Versions of Existing Test Cases

Joel David Martin, Ph.D.

The University of Texas at Arlington, 2019

Supervising Professors: Farhad Kamangar and David Levine

Verifying that a web browser rendering engine correctly renders all valid web pages is challenging due to the size of the input space (valid web pages), the difficulty of determining correct rendering for any given web page (the test oracle problem), and the degree to which normal variation in browser rendering behavior can obscure other differences (fonts, borders, input controls, etc). These challenges lead to manual human involvement during the testing process. We propose a new Property-Based Testing (PBT) approach that addresses these challenges in order to enable automated web browser render testing. Our approach is composed of the following modules: a system for translating HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) specification data into grammar-based generators; a grammar weighting system that controls test case generation along with multiple methods for automatically adjusting those weights; and a consensus oracle of multiple rendering engines to identify failing test cases.

Our approach increases the practicality of real-world testing. It has the ability to reproduce existing test cases from external sources, is able to shrink test cases to assist with root-cause analysis, and supports generic test specifications. We validated and characterized the effectiveness of our approach with a constrained markup grammar developed for this purpose. Applying our approach while testing Mozilla Firefox and Google Chrome revealed 1695 unique test cases that yield rendering differences. Testing of Mozilla's new Servo web browser revealed multiple bugs including eleven crashes and resource leaks. We reported these issues to Mozilla developers and are working together to develop and verify solutions.

Contents

Listings

List of Figures

List of Tables

Acronyms and Abbreviations Used

AST	abstract syntax tree. 96, 128
BNF	Backus-Naur form. viii, xiv, 18, 101, 102
CFG	Context-Free Grammar. 197
CSS	Cascading Style Sheets. vi, viii, xiv, xv, xx, xxiii, xxiv, 1–3, 6, 7, 9, 12, 15, 20, 24, 26, 29, 34, 59, 101, 107–121, 124, 125, 128, 130, 131, 134, 135, 140, 144–148, 159, 163, 164, 166, 171, 173, 174, 179–181, 184, 193–195, 197, 198, 208, 212, 214–216, 218, 220, 221
CSS3	Cascading Style Sheet Level 3. viii, 1, 12, 16, 18, 20–22, 24, 27, 101, 107, 110, 113
DAG	directed acyclic graph. 36
DOM	document object model. 125, 192
DSL	domain-specific language. 195
EBNF	Extended Backus-Naur form. viii–x, xii–xvi, xviii–xx, xxiii, 6, 7, 9, 16, 18–20, 29, 34–36, 45, 59, 71, 72, 75, 76, 82–102, 105, 106, 110–113, 116, 117, 119–121, 123, 124, 128, 130–133, 173, 174, 194, 195, 197–203, 208–210
FSM	finite state machine. 192

HTML	HyperText Markup Language. vi, viii, xiv, xv, xx, xxiii, xxiv, 2, 3, 6, 7, 9, 15, 18–20, 24, 26, 29, 32, 34, 59, 101, 103–110, 117–123, 128, 130–135, 139, 140, 142, 145–147, 159, 163, 164, 166, 171, 173, 174, 179, 181, 191, 193–195, 198, 212, 213, 216, 217, 219–221, 227
HTML5	Hypertext Markup Language version 5. viii, 1, 12, 16, 18–22, 24, 27, 101, 103, 139
HTTP	HyperText Transfer Protocol. 157, 158
IE7	Internet Explorer 7. 190
IR	intermediate representation. 186
JS	JavaScript. 157, 158
JSON	JavaScript object notation. 14, 113
MDN	Mozilla Developer Network. xx, 113–116, 128
PBT	Property-Based Testing. vi–viii, x, 2–6, 9, 12–15, 27, 29–34, 37, 42, 45, 62, 63, 86, 89, 92, 94, 131, 134, 136, 146, 169, 185, 186, 190, 191, 194, 195, 197
PEG	parsing expression grammar. 62, 198
PNG	Portable Network Graphics. 16, 128
PRN	pseudo random number. 31–33
SSD	sum of square of differences. 23, 59, 60, 136, 145, 175
SUT	system under test. xiii, 2, 12, 13, 29–34, 37, 46, 51, 63, 89–91, 94, 95, 135, 136, 190, 194–196
SVG	Scalable Vector Graphics. 125

TAPV	combination of tag, attribute, property, and property name. xii, xviii, 45–47, 51–57
UML	Unified Modelling language. 191
URL	Uniform Resource Locator. 157, 158
VDS	Value Definition Syntax. viii, xiv, 20, 110–113, 116, 128, 197, 208
VLSI	very large-scale integration. 188
W3C	World Wide Web Consortium. xiv, 59, 103–107, 113, 117, 119, 120, 127, 128, 157, 194
WaSP	Web Standards Project. 125, 127
WHATWG	Web Hypertext Application Technology Working Group. 103, 139
WPT	web-platform-tests. 127, 198
XBIS	cross-browser incompatibilities. 192
XHTML	Extensible HyperText Markup Language. 103
XML	Extensible Markup Language. 103
YAML	Yet Another Markup Language. 158, 169

Chapter 1

Introduction

1.1 Motivation and Objectives

1.1.1 Browser Testing Challenges

One of the challenges of developing a new browser engine is in creating a rendering engine with correct rendering behavior across the entire range and possible combinations of page elements and layouts. The HTML5 [2] and CSS3 [3] standards consist of large and complicated specifications. Verifying that a browser rendering engine correctly renders all possible pages is a significant testing challenge.

One way to approach this challenge is to manually create test cases with corresponding reference images and then compare the resulting rendered page to the reference image. This is the method that the Acid Tests [4] use to test various CSS standard versions. This method involves manual creation of both the test cases and the reference images.

Another approach to browser render testing is to create two different test pages that should both have the same visual appearance when rendered but which use different underlying mechanisms to achieve it. Differences in the results indicate a possible rendering error/defect. It is possible to automate test case creation, however creating a model that is able to determine when two separate cases should result in the same rendering can still present a significant challenge. In addition, the type of test cases that can be tested with this method is limited to those types of pages and layouts that have multiple rendering paths with the same visual result. There is also likely to be significant overlap in the methods used within a test case to achieve the same visual rendering. Some types of defects within these common paths can result in renderings that are the same in both cases but incorrect nonetheless.

The principal problem with these approaches is that human intervention is necessary during input creation or output validation or both. We propose a technique that uses Property-Based Testing (PBT) to automate both the creation of inputs (web page test cases) and the validation of rendered output. This is achieved by automated input generation using grammar-based test generators and automated output validation using a consensus oracle.

Property-based testing (sometimes referred to as "generative testing") is a method of automated software verification that can reveal software defects beyond what traditional hand-written tests can reveal. Tests are defined in terms of input and output properties. The input properties are used by the test system to generate the test cases (often stochastically). The tests are run against the SUT and the results are validated using the output properties. Test cases which violate the output properties are defects in the SUT.

However, this simple description belies the fact that creating input and output properties for non-trivial software can present significant challenges. Input properties might need to specify large streams of complex data. Output properties present a larger challenge because they often need to be a fairly complete model of correct behavior for the SUT. This testing model is known as a test oracle and developing a test oracle for a complex piece of software can approach the complexity of the original system. The "test oracle problem" is why property-based testing is often reserved to unit testing where it is simpler to infer correct behavior.

Browser rendering is an area that could greatly benefit from fully automated property-based testing. However, generating interesting inputs (HTML and CSS) and creating a test oracle that can detect an incorrectly rendered web page based on the inputs is a significant challenge. Another significant challenge is the fact that current popular web browsers have significantly divergent “normal” behavior. In this paper, we propose a new PBT approach that addresses these challenges so that web browser render testing can be performed with minimal human intervention in the process. We have successfully used the technique to identify nearly 1700 unique web page test cases that are rendered differently between the Mozilla Firefox and Google Chrome rendering engines.

1.1.2 Solving the Limitations of PBT with Instacheck

PBT is a powerful automated testing approach and we have had success using it in an industry setting to discover bugs that went undetected using standard example-based testing methods. This was strong motivation for us to use it as the basis for automated testing of browser rendering engines. However, this testing domain is different from the typical use of PBT due to the size and recursive nature of the input space (HTML and CSS web pages); and due to the variability of browser rendering behavior (large number of counterexamples). We identified the following limitations with the standard PBT approach when we attempted to apply it to browser render testing: input property generators are not defined using a general grammar language; adjusting test coverage involves updating or creating new test input property generators; and the same test cases tend to be rediscovered especially when the PBT test shrinking capability is used.

We developed a new approach to PBT, Instacheck, that addresses these limitations and also enables a new PBT capability: the ability to reproduce and shrink of preexisting test cases. Although we developed it to address the issues we discovered during browser render testing, Instacheck is a generic PBT-based approach and can be applied to many other testing contexts.

1.1.3 Use case: a New Web Browser Engine

The Mozilla foundation started the Servo project in 2012 [5] with the goal of developing a web browser engine with improved parallelism, security, modularity, and performance [6]. The project is using the Rust language [7] to build the Servo browser and rendering engine. New components are being developed for all aspects of web page rendering including: parsing, layout, image decoding, tile rendering, compositing, etc [8]. We believe that there is significant opportunity for fully automated web browser render testing to accelerate the development of new web browsers like Servo. The use of this system to advance the development of the Servo engine is one of the motivations for this work. Using the system we identified over 250 unique web pages that Servo renders differently from Firefox and Chrome. In addition we discovered 11 confirmed bugs in Servo including several crashes due to resource leaks. We have worked with Mozilla engineers to resolve several of these bugs and are working to resolve the remaining ones. We were also able to reproduce several preexisting Servo bugs using our technique and then leverage the shrinking capability of PBT to shrink those bugs into simpler test cases.

1.1.4 Objectives

The objectives for this research fall into three general categories which are listed below along with related sub-objectives. Each item references the sections of the document that demonstrate how the objectives were satisfied.

1. **End-to-end automated browser render testing**

   (a) Run a large number of tests to show that the system is able to effectively detect browser render differences. Determine the degree to which the test cases are able to be shrunk by the PBT shrinking functionality (7).

   (b) Support testing of any unmodified web browser that supports the WebDriver testing framework. In particular, browsers should not require code changes or special instrumentation in order to be validated. In addition, support the use of online cloud testing services such as BrowserStack 6.4.

   (c) Perform targeted testing of Mozilla's Servo browser and report any bugs discovered (7.2).

2. **A grammar weight system that increases test case coverage through automatic weight tuning and that can reproduce and shrink existing test cases**.

   (a) Develop a constrained web page-like grammar with synthetic error conditions that can be used to empirically validate various aspects of the Instacheck system (3.5.1).

   (b) Reduce the probability of generating the same failing test cases repeatedly by using automatic weight tuning algorithms between test runs (3.4.4). Validate this capability using the constrained grammars 3.5.3). This is especially important for browser render testing because web browsers have divergent rendering behavior that is consider "normal", but this divergent behavior can hide other real bugs.

   (c) Generate new failing test cases that are similar to but smaller and/or simpler than existing failing test cases (3.4.5).

   (d) Use the constrained grammar to characterize how well the system is able to reproduce test cases with various amounts of noise in the test case (3.5.5). The ability of PBT to shrink test cases to more minimal test cases is very powerful and the grammar weighting technique enables this capability to be used ala carte with existing test cases.

   (e) In order to support arbitrary web page test cases, make sure that the translated HTML and CSS EBNF grammars are complete enough to parse normal valid web pages (5.4).

3. **Practical utility for testing**. There are a number of software testing approaches which appear powerful in theory but end up being narrowly applicable or difficult to use for practical testing. We identified a number of specific capabilities to make this research practically useful including:

   (a) A modular architecture that enables components of the system to be separately useful (1.2, 4.5, 5.5, 6.6).

   (b) Provide testers with multiple levels of controllability including the grammar generator weights, choice of automated selection/reduction algorithms, and configurable consensus decision algorithms (and G).

   (c) Command line PBT usage of Instacheck (4.5), html5-css3-ebnf (5.5), and Bartender (6.6). This includes the ability to reproduce and shrink existing test cases with both Instacheck and Bartender.

1.2 Modules and Document Structure

1.2.1 Modules

Our approach to achieving the objectives consists of three complementary modules. Figure 1.1 shows the full system architecture with each module outlined. The modules are described below with references where the modules are described in this document and the URLs where the modules are published as open source projects.

- **Instacheck**: `https://github.com/kanaka/instacheck`

  A property-based testing approach with test specifications defined generically as EBNF, with a weighted grammar model that controls test case generation, and with with multiple methods for automatically adjusting grammar weights to either avoid already discovered test cases or reproduce existing test cases (chapters 3 and 4). Instacheck is outlined with a blue dotted box in Figure 1.1.

- **html5-css3-ebnf**: `https://github.com/kanaka/html5-css3-ebnf`

  A method for parsing and translating HTML and CSS specification data into EBNF grammars (Chapter 5). html5-css3-ebnf is outlined with a green dotted box in Figure 1.1.

- **Bartender** `https://github.com/kanaka/bartender`

  (**B**rowser **A**utomated **R**ender **T**esti**N**g **D**riv**ER**): An automated end-to-end browser render testing architecture that uses Instacheck and html5-css3-ebnf along with a web browser consensus oracle to identify failing test cases (Chapter 6). Bartender is outlined with a red dotted box in Figure 1.1.

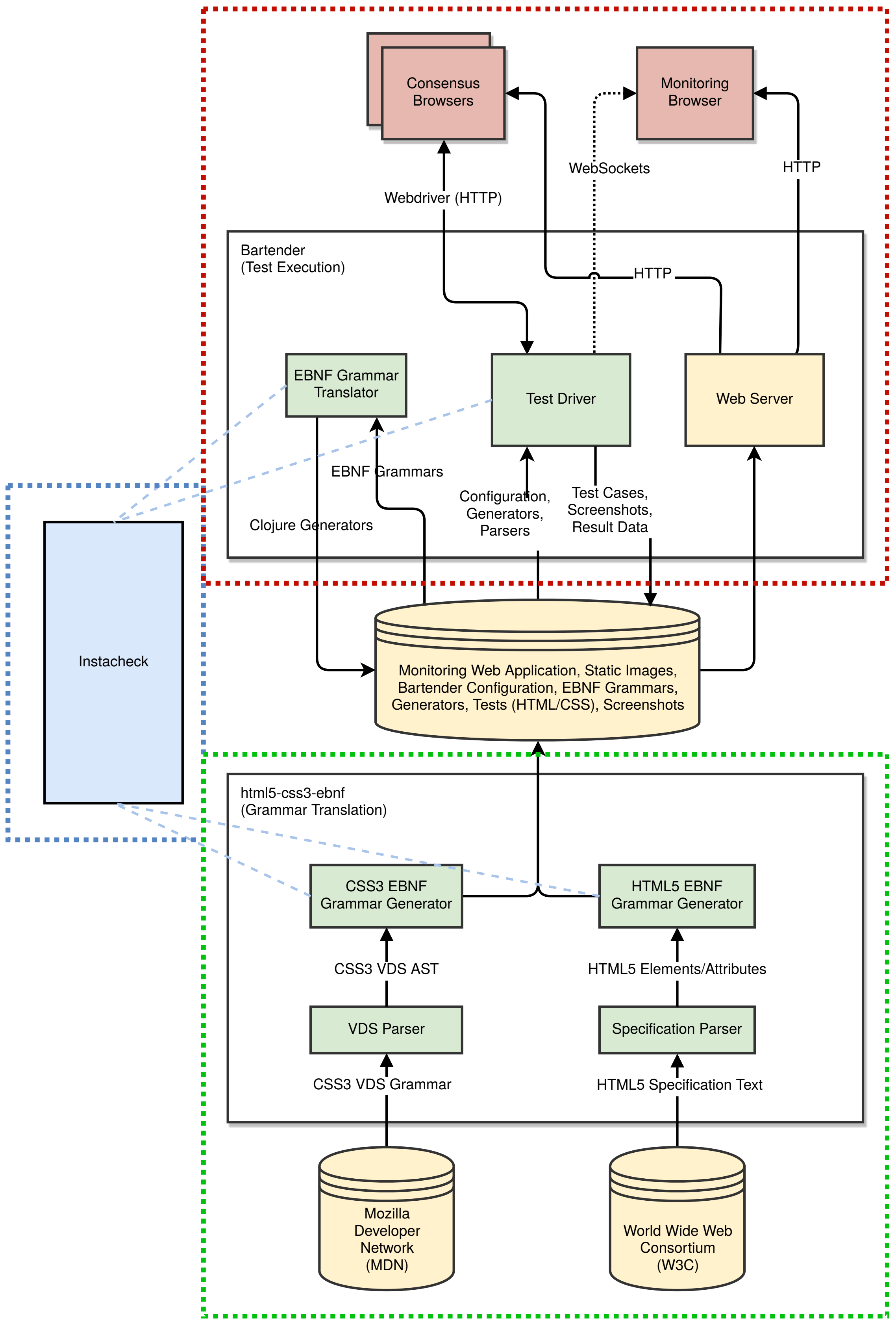

Consensus Browsers
Monitoring Browser
WebSockets
HTTP
Webdriver (HTTP)
Bartender (Test Execution)
HTTP
EBNF Grammar Translator
Test Driver
Web Server
EBNF Grammars
Configuration, Generators, Parsers
Test Cases, Screenshots, Result Data
Clojure Generators
Instacheck
Monitoring Web Application, Static Images, Bartender Configuration, EBNF Grammars, Generators, Tests (HTML/CSS), Screenshots
html5-css3-ebnf (Grammar Translation)
CSS3 EBNF Grammar Generator
HTML5 EBNF Grammar Generator
CSS3 VDS AST
HTML5 Elements/Attributes
VDS Parser
Specification Parser
CSS3 VDS Grammar
HTML5 Specification Text
Mozilla Developer Network (MDN)
World Wide Web Consortium (W3C)

Figure 1.1: Test System Architecture With Highlighted Modules

1.2.2 Document Structure

This document is structured as follows:

- Chapter 2 was published as **"Property-Based Testing of Browser Rendering Engines with a Consensus Oracle"** (COMPSAC 2018) [1]. This serves as a high-level overview of PBT, the translation of HTML and CSS specification to EBNF grammars, controllable weighted grammars (later named Instacheck), consensus oracle, and an architecture that combines all those components to enable end-to-end automation of web browser render testing.

- Chapter 3 introduces the Instacheck technique. This includes the use of EBNF grammars for defining PBT input properties, a weighted grammar for controlling test coverage, multiple weight reduction algorithms for either reducing the rediscovery bugs or for reproducing existing external test cases.

- Chapter 4 provides greater depth about the PBT, Clojure's *test.check* library, and the **Instacheck** technique. Practical use of Instacheck as a library and as a command line tool is also described.

- Chapter 5 describes the **html5-css3-ebnf** module and the process and challenges of sanitizing and translating HTML and CSS specification data into EBNF grammars that are useful for both parsing web pages and for creating random web page generators. Practical use of **html5-css3-ebnf** as a command line tool is also described.

- Chapter 6 covers the architecture of the end-to-end system as defined by the **Bartender** module. **Bartender** combines html5-css3-ebnf, Instacheck, and a consensus oracle to enable fully automated web browser render testing. Practical use of **Bartender** as a command line tool is also described.

- Chapter 7 describes the results of using **Bartender** to test two different consensus pools and consensus algorithm configurations. The first configuration compares Mozilla

Firefox to Google Chrome. The second configuration uses a combination of Firefox and Chrome to do targeted testing of Mozilla Servo. This chapter expands on the browser render testing use-case introduced in Chapter 3.

- Chapter 8 describes other research and literature that is related to, or provides a foundation for, the research described in this document.
- Chapter 9 has concluding remarks and discussion of future work that should be pursued.

Chapter 2

“Property-Based Testing of Browser Rendering Engines with a Consensus Oracle” (COMPSAC, July 2018)

Authors:

Joel Martin, University of Texas at Arlington, joel@martintribe.org

David Levine, University of Texas at Arlington, levine@cse.uta.edu

2.1 Abstract

Verifying that a browser rendering engine correctly renders all valid web pages is a challenging problem due to the size of the input space (valid web pages) and the challenge of knowing whether the rendering for any given page is correct (the test Oracle problem). We propose a PBT approach that uses controllable grammar-based generators for creating the inputs (web pages containing HTML5 and CSS3) and that uses a (lack of) consensus among multiple rendering engines to identify failing test cases. To assist with root-cause analysis the system uses test shrinking to report much smaller versions of failing test cases. This technique may also be used to test other software systems where there are multiple preexisting implementations of the SUT.

2.2 Introduction

One of the challenges of developing a new browser engine is achieving a rendering engine with correct rendering across the wide range of possible combinations of page elements and layouts. The HTML5 [2] and CSS3 [9] standards consist of large and complicated specifications. Verifying that a browser rendering engine correctly renders all possible pages is a significant testing challenge.

One way to approach this challenge is to manually create test cases with corresponding reference images and then compare the resulting rendered page to the reference image. This is the method that the Acid Tests [4] use to test various CSS standard versions. This method involves manual creation of both the test cases and the reference images. Another approach is to create two test pages that have the same rendered visual appearance but which use different underlying mechanisms to achieve it. Differences in the results indicate a possible rendering error/defect. The type of test cases that can be tested with this method is limited to those types of pages and layouts that have multiple rendering paths with the same visual result. While it is possible to automate test case creation, creating a model that can generate web pages which result in the same rendering using different page elements is a significant challenge.

The key problem with these approaches is that the model (Oracle) used to validate a test case is either via manual human inspection or the process of creating tests is manual, complex and limited in scope. In other words, human intervention is necessary during input creation or output validation or both. To address these challenges we propose a PBT system (Section 3.3) that automates both the creation of inputs (web page test cases) and the validation of rendered output. This is achieved by automated input generation using grammar-based test generators (Section 2.5 and Section 2.6) and automated output validation using a consensus Oracle (Section 2.7). The system architecture and the results of using the system are described in sections 2.4 and 2.8 respectively.

2.3 Property-Based Testing (PBT)

PBT (also known as generative testing) is a method of automated software verification in which tests are defined in terms of input and output properties. This is in contrast with the common example-based testing in which each test is defined as a fixed set of inputs, the mechanism for running the tests, and the expected outputs for that test.

In PBT the possible inputs to the SUT are defined as properties. These properties are used by the test system to generate valid inputs for the SUT that satisfy the input properties. Using the input properties to generate a comprehensive set of all possible inputs is an option if the input property search space is relatively small. However, this is rarely a tractable option: a program that takes two 4-byte integers has 1.84×10^{19} possible inputs without any further constraints. For this reason, PBT systems often generate the inputs stochastically. In addition to the input properties, the developer defines output properties which are used to determine if the given input results in a correct outcome when the test is run.

One of the most popular PBT systems is QuickCheck [10] which was originally developed in Haskell but has been ported to many other languages. Our system is written in Clojure [11] and leverages Clojure's *test.check* library [12] which is a PBT library based on QuickCheck.

2.3.1 Test Generation using Clojure test.check

The *test.check* library provides a number of base generators. For example, Listing 2.1 shows a generator that generates a random integer. In the following listings *tc*, *gen*, and *prop* are abbreviations for the *test.check*, *test.check.generators*, and *test.check.properties* namespaces respectively.

```
gen/int
```

Listing 2.1: Basic Generator

Generators can be combined into compound generators. For example, the following generator outputs a two element tuple that contains a random integer followed by a random alphanumeric string:

```
(gen/tuple gen/int gen/string-alphanumeric)
```

Listing 2.2: Compound Generator - Tuple

A more interesting case is the following generator which outputs a random length vector containing random integers:

```
(gen/vector gen/int)
```

Listing 2.3: Compound Generator - Vector

Recursive generators can also be defined. For example, the following will generate JavaScript object notation (JSON) like structures containing a random hierarchy of maps and lists:

```
(defn compound [inner-gen]
  (gen/one-of [(gen/list inner-gen)
               (gen/map inner-gen inner-gen)]))

(def scalars
  (gen/one-of [gen/int gen/boolean]))

(def json-like
  (gen/recursive-gen compound scalars))
```

Listing 2.4: Recursive Generator

Test generators can be thought of as tree data structures in which the leaves of the tree are the fundamental generators (i.e. provided by *test.check*) and the internal nodes of the tree are compound generators that are defined in terms of fundamental generator leaves. In the case of the browser test system, the root node of the tree is a compound generator that generates full web pages (HTML/CSS) and the leaves of the tree are fundamental generators that generate atomic elements of the page such as HTML tags or CSS property names.

When a generator (fundamental or compound) is invoked it takes two parameters: a sizing value and a random seed value. Invoking the same generator again with the same size and seed results in the same generated result. Note that the size value has a meaning that is specific to each type of generator. For example, for a random integer generator, the size determines the maximum possible magnitude of generated integer value. For example if *gen/int* is invoked with a size of 10 then it will output a random value between -10 and 10. In the case of *gen/vector* the size value determines the maximum length of the vector. During normal test generation mode, compound generators invoke their child generators using their own size value and random seed values that are deterministically derived from their own size and random seed value.

2.3.2 Test Shrinking

Most QuickCheck derived PBT systems provide a feature called test shrinking that attempts to identify the smallest version of the test case that continues to fail (does not satisfy the test property).

A failing test case is defined in terms of a generator tree with the size and random seed values that were used at every node and leaf of the tree. The shrinking process modifies the current instantiated generator tree by picking a random node in the current instantiated generator tree and replacing the size value with a smaller size value. This new instantiated generator tree is used to generate a new test and if that test continues to fail the system will adopt this as the current smallest test case. If the test passes the system will backtrack and attempt to shrink a different node of the instantiated generator tree.

2.4 Test System Architecture

Fig. 2.1 shows the basic architecture of the testing system. A test case (complete web page) is randomly generated (Fig. 2.1-H) from the combined HTML5 and CSS3 EBNF grammar (Section 2.5) based on the current seed and size setting. The test driver uses the WebDriver protocol [13] to establish a connection (Fig. 2.1-J) to each browser in the consensus pool (Fig. 2.1-K) and requests that the test page (including any referenced static resources) is loaded via the web server (Fig. 2.1-G).

In parallel, each browser loads and renders the test page (Fig. 2.1-L). Then the test driver requests that each browser take a screenshot of the rendered page and return it as a Portable Network Graphics (PNG) image to the test driver (Fig. 2.1-M). For each set of browser screenshots, the test driver determines if there is a consensus rendering (Section 2.7.2).

The process is repeated with new test cases until the number of passing tests reaches the configured number of test cases or until the first failing test is found. Once a failing test case is found, the test driver switches to test shrink mode and attempts to find the smallest and/or simplest test case that still fails (Section 2.3.2).

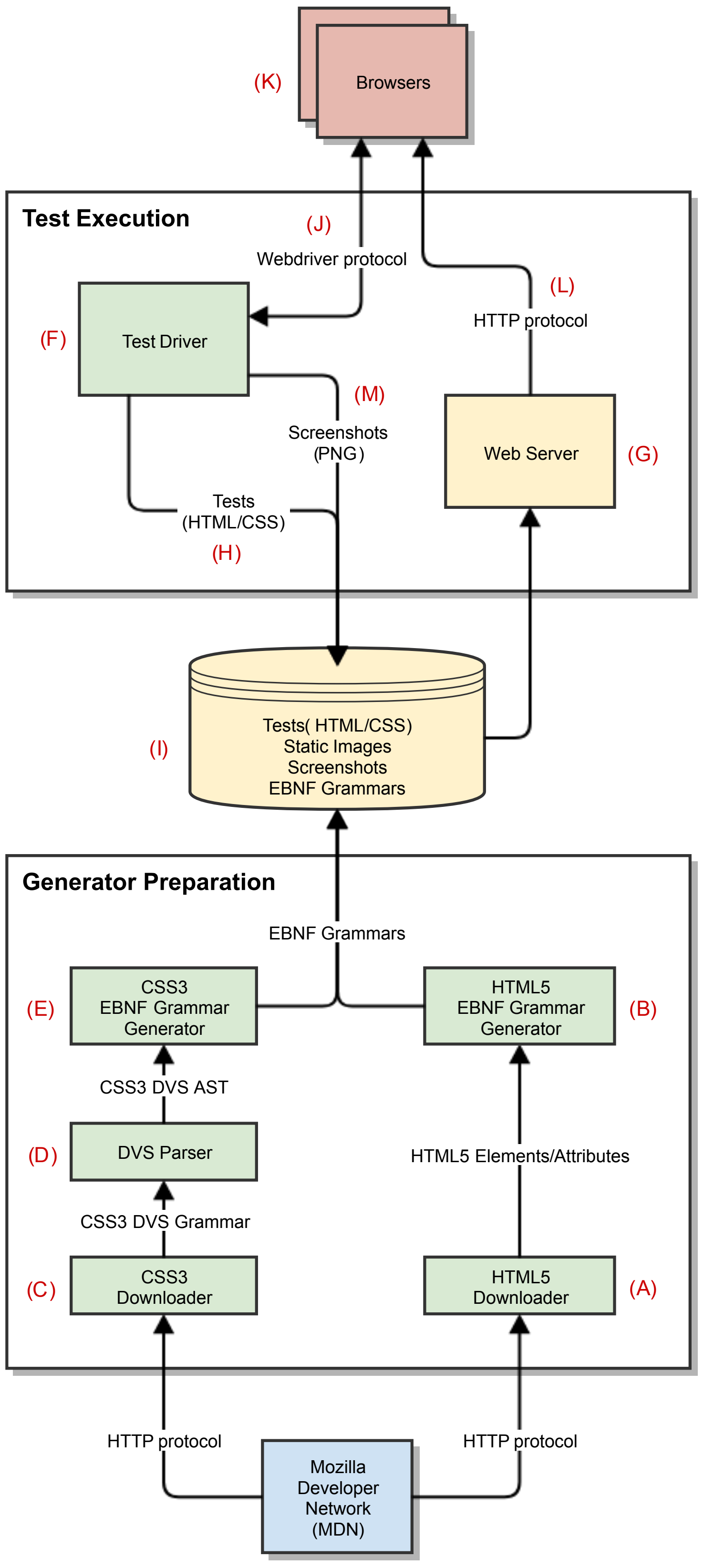


Figure 2.1: Test System Architecture

2.5 Grammar-based Input Generators

One option for generating web page input would be to manually write *test.check* generators for generating HTML5 and CSS3. However, given the large scope of the HTML5 and CSS3 standards a more automatic solution is necessary to get wide test coverage. Since both standards can be described in terms of formal grammars, we propose a method that downloads, scrubs and compiles the HTML5 and CSS3 specification data into Extended Backus-Naur form (EBNF) grammars. Backus-Naur form (BNF) is a formal notation for describing context-free grammars. EBNF is the name for a number of variants of the original BNF with increased expressive power by adding features like grouping, repetition, and regular expressions. Our test system uses the *instaparse* [14] Clojure library which supports most of the popular EBNF variant syntaxes. Table 2.1 contains a summary of the EBNF expansion syntax supported by *instaparse*.

EBNF grammars are used in our system as a common intermediate format that is then translated into *test.check* generators that can be loaded and executed by the test system. These *test.check* generators are used during browser render testing. The EBNF translation and *test.check* execution components of our system are generic and can used to test any command-line application for which the inputs are (or can be) defined with an EBNF grammar.

2.5.1 HTML5 Grammar

HyperText Markup Language (HTML) is a format for describing the content and semantic layout of a web page. A web page is a structured hierarchy of HTML elements which are delineated by tags. Normal HTML elements are defined with a start and end tag. A start tag is a string containing the element name surrounded by angle brackets such as <*div*>. An end tag is similar to a start tag but the element name is prefixed with a slash such as </*div*>. The content of the tag is defined between the start and end tags and may contain

Table 2.1: EBNF Expansions

`A B` or `A, B`	concatenation
`A | B`	alternation
`(A B)`	grouping
`A?` or `[A]`	optional (zero or one)
`A+`	one or more
`A*` or `{A}`	zero or more
`"abc"` or `'abc'`	string literal
`#"abc"` or `#'abc'`	regular expression

textual data and definitions for other elements.

The start tag for HTML elements may contain element attributes after the tag name and before the ending angle bracket. These are enumerated as key/value pairs with a = symbol between the key and value. The attribute value is typically surrounded by double-quotes although this is only required when the value contains spaces.

The structural grammar for HTML5 is fairly shallow even though it contains over 130 tag elements and 110 tag attributes. In theory, we could just define these in terms of any string of valid characters as shown in this EBNF grammar snippet:

```
element = '<' tag-char+ (<space> attribute)* '>'
          (element | content)* '</' tag-char+ '>'
attribute = attr-char+ '="' char-data '"'
```

Listing 2.5: Simple EBNF

The grammar can parse structurally sound web pages but there are two key problems that arise when we attempt to use this grammar to generate random web pages. The first problem is that because EBNF is context-free we don't have a mechanism to generate arbitrary matching start and end tags. Even if we had way to address this problem (e.g. using a non-context free grammar) we would still have a second problem. Since the generated tag names are random strings it is exceedingly rare that the strings will match HTML5 tags and trigger interesting structural or rendering semantics in the browser. For these reasons, the EBNF grammar that we generate (Fig. 2.1-B) contains full definitions for HTML5 tags and attributes. Listing 2.6 shows a snippet of the resulting HTML5 EBNF grammar.

```
element = '<a' (<space> a-attribute)* '>'
            (element | content)* '</a>'
          | '<abbr' (<space> abbr-attribute)* '>'
            (element | content)* '</abbr>'
a-attribute = 'download="' attr-val-download '"'
              | 'href="' attr-val-href '"'
```

Listing 2.6: HTML5 EBNF

2.5.2 CSS3 Grammar

Cascading Style Sheet Level 3 (CSS3) is a language for describing the presentation or visual appearance of the HTML element of a web page. CSS properties modify the appearance or behavior of HTML elements. Each property is composed of a property name followed by a colon followed by a one or more property values. Listing 2.7 shows a normal *div* tag that has a style attribute containing contains some text. The *div* element has single style attribute containing two CSS properties which adjust the font size and color of any textual content within the *div* element.

```
<div style="font: 2em; color: red">
  my test
</div>
```

Listing 2.7: Simple Div

The CSS3 standard is more expansive than the HTML5 standard and consists of over 50 different sub-specifications referred to as CSS3 modules which are in various stages of standardization [15]. There are more than 360 CSS property names across all the CSS3 modules. The CSS3 Property Values are defined in terms of a formal grammar called Value Definition Syntax (VDS) [16]. Th VDS grammar is similar to EBNF but has some additional combinators and multipliers that allow concise definitions of common patterns in CSS property values. In addition whitespace between terminal elements is implicit.

We define a separate EBNF grammar specifically for parsing VDS data so that we can reuse our parsing code (Fig. 2.1-D). The parsed VDS data is then translated into the final EBNF representation (Fig. 2.1-E) for use in the test execution system. The following listings shows the VDS for the *text-emphasis-position* data value and the translated EBNF grammar:

```
<'text-emphasis-position'> = [ over | under ] &&
                             [ right | left ]
```

Listing 2.8: “text-emphasis-position” VDS

```
prop-text-emphasis-position =
  (
    ( 'over' ' ' | 'under' ' ') ' '
    ( 'right' ' ' | 'left' ' ') ' '
  ) ;
```

Listing 2.9: “text-emphasis-position” EBNF

2.6 Generator Tuning

During test execution, a weights file can be used to tune the behavior of the HTML5 and CSS3 generators. The weights are used to adjust the frequency distribution for alternation rules in the grammar. The default weights are set to the same value for all components which results in a uniform distribution (each alternation component will be selected with an even probability). More fine-grained grammar control is discussed in Lämmel 2006 [17]. There are a number of scenarios where tuning of generator weights is useful including:

1. **Single component testing**: The weights can be tuned to focus test generation on one component (or a small number of components) by significantly increasing the weights along the path to the component of interest in the grammar.

2. **Avoid known failures**: Failures that occur at smaller test case sizes can mask failures at larger sizes. The weights can be tuned to avoid the components causing the known failures by setting the weight to zero (or much smaller values). This will increase the likelihood of discovering unique failures.

3. **Pairwise or combinatorial testing**: Empirical studies (Appendix B of Kuhn 2010 [18]) show that a high percentage of software failures are caused by the interaction of a small number of variables and the interaction of just two variables is typically

sufficient to discover over 50% of software failures. Pairwise or combinatorial testing can be performed using our test system by generating weight files that have pairs or combinations of weights with large values (high probability).

4. **Typical and atypical test cases**: An existing collection of web pages can be used to tune the test system. For example, if a tester wants to gain confidence that popular web pages will render correctly, a collection of web pages could be created of the most frequently visited web pages. The existing HTML5 and CSS3 grammars can be leveraged to parse these web pages to determine the average frequency with which different components appear and this frequency data can be used to set the generator weights. Alternatively if the tester is interested in uncommon HTML5 and CSS3 components then the weights can be set to an inverted distribution based on the frequencies.

2.7 Consensus

2.7.1 Challenges

In a consensus Oracle testing model (named "differential testing" in McKeeman 1998 [19]), test cases are loaded in multiple different implementations and if there is a consensus then the current test case is considered to be passing. The challenge with a consensus Oracle is identifying false positives: differences that do not indicate a true failure. The web standards provide some leeway in terms of how browsers are allowed to render the same test case so it is important to either eliminate these differences from the generated test cases or to identify and ignore differences that do not indicate a failure. We have identified element borders [20] and font rendering as the two most problematic areas.

The most obvious solution to permissible rendering differences is to avoid these cases (at least initially). This is fairly simple for the element border differences because the test generators can simply be configured to not use problematic border features. However, font rendering is more likely to reveal rendering defects in a browser and omitting fonts from testing would result in a significant testing gap. The Ahem font [21] was designed specifically for the purpose of browser testing. The Ahem font consists of filled squares and rectangles that are precisely specified so that they are able to be rendered the same regardless of platform and rendering engine [22].

2.7.2 Calculating Consensus

We define consensus for a single input test case as the condition where all pairs of rendered images have a disagreement value that is less than a threshold. Given a set of browser renderings B for a single input test case, an algorithm R that gives a measure of disagreement between two rendered images a and b, and a maximum disagreement threshold θ for that algorithm, then consensus is when Δ is the empty set:

$$\Delta = \{\{a, b\} \in B \times B \; | R(a, b) >= \theta\}$$

Conversely if Δ is not the empty set then there is not a consensus. The algorithm that is used for calculating the disagreement value is a configurable option in the system but the default algorithm is to calculate a normalized sum of square of differences (SSD) as follows:

$$R(a, b) = \frac{\sum_{x,y} (a(x, y) - b(x, y))^2}{\sqrt{\sum_{x,y} (a(x, y)^2 . \sum_{x,y} b(x, y))^2}}$$

Additionally, we can define a particular browser x as being at fault for the disagreement if every browser-pair containing x is also in Δ (the browser disagrees with every other browser).

$$\{\{x, b\} \in x \times B \; | x \neq b\} \subset \Delta$$

Note that more than one browser may be at fault since more than one browser may be in disagreement with all other browsers. This includes the case where all browsers are fault.

There is also the interesting case where there is disagreement among the browsers but no particular browser is identified as at fault. Consider an Oracle composed of three browsers x, y and z where $R(x, z) >= \theta$ (disagreement) but $R(x, y) < \theta$ and $R(y, z) < \theta$. In other words x and z are in disagreement but y has a rendering that has split the difference and has feature of the others such that neither $R(x, y)$ or $R(y, z)$ are in disagreement. It is not clear in this case whether the fault lies with x or z or perhaps all three.

There are alternatives for calculating which browser is most at fault for a disagreement in the case where no single browser disagrees with all other browsers. One method would be to generate an average image of all the rendered images and then identify which browser is most different from the average. However this would conflate two different types of disagreement and we found it useful to be able to distinguish between all four categories of agreement/disagreement: all browsers in agreement, disagreement with one browser most at fault, disagreement with all browsers at fault, disagreement with no single browser at fault.

2.8 Results

To validate the capabilities of our system we used a configuration that included three browsers in the consensus pool: PhantomJS (version 2.1.1), Mozilla Firefox (version 52), and Mozilla Servo (release build of git hash 854d720b2). Our initial tests used the full HTML5 and CSS3 grammar including elements that are marked obsolete and experimental in the standards. These elements are less standardized across browsers and so we expected to easily find browser differences due to these elements being included in test cases. We also used a minimally tuned weight file to focus the test cases on visual rendering differences.

Step 1 of each test run uses the smallest test case that can be generated by the grammar. The HTML/CSS of this test case (shown in Listing 2.11) is an *html* element containing a single *body* element with no content. The *body* element has a default style that specifies a

```
<html>
  <body style="background: #1289af;
               font: 25px/1 Ahem">
    <marquee bgcolor="navy">
      <q cite="STUB_cite" ...ELIDED... >
        <!---uo--><!--WZ-G-G--><!--iM--> <!---->pX
      </q>
    </marquee>
    p
    <mark ...ELIDED...> </mark>
    <strong ...ELIDED...
            style="offset-anchor: right;
                   box-align: stretch;
                   padding-right: -1.75vw;
                   scroll-snap-type-x: mandatory">
      <!--m-->
      <mark ...ELIDED...> Xp<!---M-S-->XX </mark>
      &#x00c9;
    </strong>
  </body>
</html>
```

Listing 2.10: "text-emphasis-position" EBNF

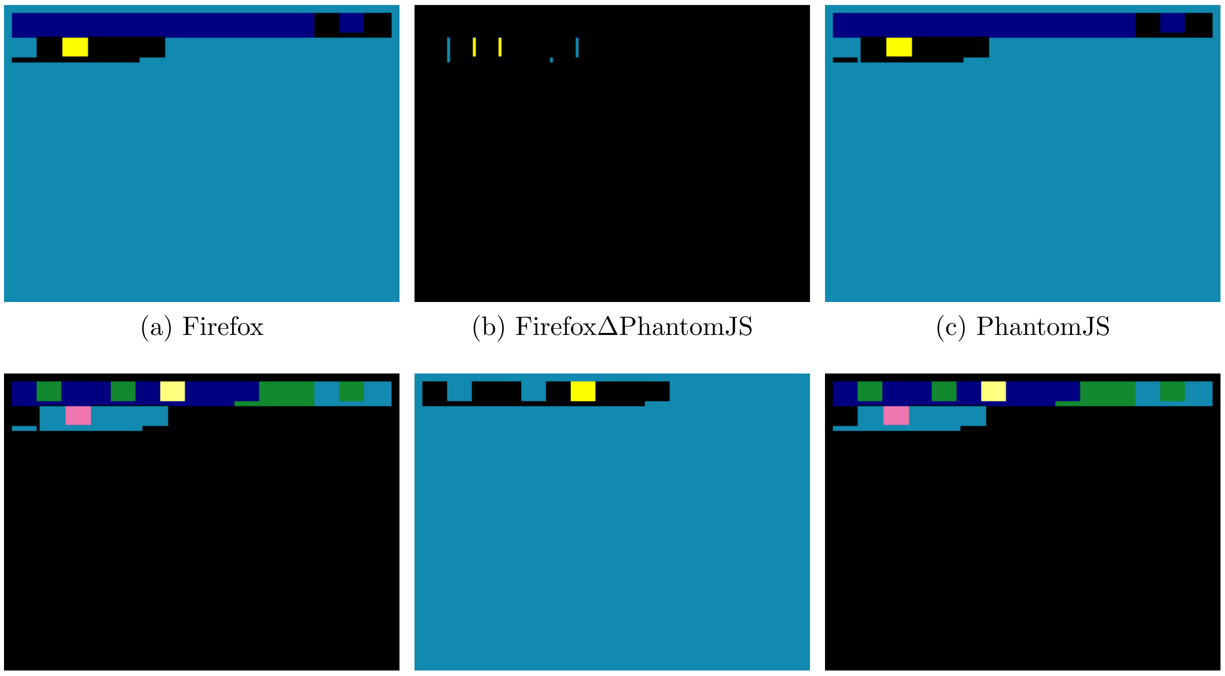

(a) Firefox (b) FirefoxΔPhantomJS (c) PhantomJS
(d) PhantomJSΔServo (e) Servo (f) FirefoxΔServo

Figure 2.2: Step 7

blue background and that the Ahem font should be used as the default font for text within the body. The reason for the blue background setting is to distinguish the case where a browser completely fails to render a particular page and just shows a default background color.

```
<html>
  <body style="background: #1289af;
               font: 25px/1 Ahem">
  </body>
</html>
```

Listing 2.11: Smallest Test Case

Fig. 2.2 shows expanded results for the first step where the output property is violated (step 7). The HTML/CSS code that was generated is shown side-by-side with browser screenshots and browser difference images. The differences between the screenshots can be seen in sub-figures b, d and f of fig. 2.2. Just from visual inspection it can be seen that the difference between Firefox and PhantomJS is much less that the difference between Servo and the other two browsers. However, the difference between Firefox and PhantomJS still resulted in a value that was over the threshold resulting in all three browsers marked as at fault.

Once the test system finds a test case that violates the output property (consensus), it will begin searching for a smaller version of that test case which continues to violate the output property (failing test). Fig. 2.12 shows the smallest test case found that continues to violate the output property. The failure has been reduced to a single *marquee* tag that contains a single text character and has no attributes.

```
<html>
 <body style="background: #1289af;
              font: 25px/1 Ahem">
  <marquee> p </marquee>
 </body>
</html>
```

Listing 2.12: Smallest Failing Test Case

The median number of steps to identify a failing test case was 8 steps and the maximum was 15 steps. The median size of the first failing test case was 3025 bytes and the maximum was 109,983 bytes. The median number of steps needed to shrink a test case was 113. The median size of the test case after shrinking was 154 bytes and the maximum was 245 bytes.

2.9 Related Work

This section is elided because the content is subsumed by the overall Related Work in chapter 8

2.10 Conclusion and Future Work

We discovered that the system is able to quickly discover failure cases: across 130 test runs the maximum number of steps until a failing case was found was 15 (median 8). A key reason why failing cases are discovered quickly is that the PBT method grows the test case with each step. This effectively means that each step is checking many features and feature combinations simultaneously. This also means that the initial failing test case can be quite large. Therefore test shrinking is important for reducing the test case so that it is useful for human analysis. The average test case reduction was 90.05%.

The controllable grammar aspect of the system enables the targeting of specific features and reducing the rediscovery of failure cases. With manual inspection of test cases we were able to easily identify specific HTML5 and CSS3 features that might be at fault and tune the grammar to discover new test cases. The system could be extended to automatically tune the grammar to avoid already discovered failing cases or to take a failing test case from another test source or a user bug report. The grammar would be used to parse the test case and generate additional related failing test cases to assist in characterizing the nature of the failing test. This would be particularly useful for taking a large test case and generating a much smaller test case that still exhibits the failure.

Chapter 3

"Instacheck: An Automated Property-based Testing Model to Increase Coverage and Enable Directed Testing using Weighted Grammars"

Authors:

Joel Martin, University of Texas at Arlington, joel@martintribe.org

David Levine, University of Texas at Arlington, levine@cse.uta.edu

3.1 Abstract

While PBT is a powerful automated testing model, it has some significant shortcomings: test specifications are usually defined in the same programming language as the SUT rather than a more abstract and portable language; changing or increasing test coverage often requires manually updating or defining new test specifications; simple bugs can often hide the existence of more complex bugs; and the shrinking capability cannot be used with existing test cases. We propose a technique called Instacheck that addresses these issues by using EBNF defined inputs, by using an adjustable grammar weighting system to control test coverage, and by using an automatic weight-adjusting approach that can avoid known bugs across multiple runs. We use a constrained markup language to show that Instacheck addresses each of PBT's mentioned limitations. We also present a case-study that uses Instacheck to automatically identify differences between web browser rendering engines.

3.2 Introduction

PBT is a powerful automated testing approach that can reveal defects that are difficult to find with manual example-based testing [23]. Our own success with PBT in an industry setting motivated us to use it for automated testing of browser rendering engines. This testing domain is different from the typical use of PBT due to the size and recursive nature of the input space (HTML and CSS web pages); and due to the variability of browser rendering behavior (large number of counterexamples). We discovered a number of limitations affecting PBT's utility in this domain. We propose a new approach, Instacheck, that addresses these limitations and also enables a new PBT capability: the ability to reproduce and shrink of preexisting test cases.

This paper is structured as follows: an introduction to PBT and its limitations (3.3); a description of the Instacheck approach (3.4); results of validating Instacheck with a constrained grammar and with a full browser rendering use-case (3.5); related works (8); and concluding remarks (3.7).

3.3 Property-based Testing (PBT)

The canonical Property-Based Testing (PBT) system is QuickCheck [10], "a simple domain-specific language of testable specifications which the tester uses to define expected properties of functions under test. QuickCheck then checks that the properties hold in a large number of cases" [10]. The domain-specific language of QuickCheck derives from Haskell's type system. The QuickCheck model of property-based testing has implementations in other languages with less formally defined type systems but the essence of the PBT model is that inputs are generated from an input specification, and a test oracle verifies correct output/behavior of the system under test (SUT) for a given generated input [24].

Let's consider the definition of a simple SUT: $f(v) = x$. The function f takes a vector v containing a sequence of integers and floating point numbers, performs data analysis on the sequence, and returns a result x. Further, let's assume that if v contains three immediately adjacent numbers which are in increasing order, this will trigger a defect in f, resulting in an incorrect value being returned for x. To test f we first define a PBT input property for v as $V(S(I(), F()))$ where I is a scalar generator returning an integer, F is a scalar generator returning a floating point number, S is a compound generator that selects between other generators, and V is a compound generator that returns a vector of values from another generator. Listing 3.1 shows an example of vector values sampled from V.

```
[] [0] [-1 -1.0] [1.5] [-1.5 1] [4] [2] []
[3.1171875 -8 1.0 0.55078125 0.99609375 -2]
[-1 3 0 0.0 -0.2880859375] ...
```

Listing 3.1: Values sampled from the generator V

In addition to the PBT input properties, we also define a PBT output property test oracle $O(v)$ that returns the correct result for the SUT. The PBT process then checks the following assertion: for all values of v, the result of $f(v)$ must equal the result of $O(v)$.

```
{:pass? false,
 :num-tests 7,
 :fail [[0.9140625 1.9375 -6 -6 -1 2.25]],
 :shrunk {:total-nodes-visited 46,
          :smallest [[-1 0 1]]}}
```

Listing 3.2: Example result from *quick-check*

Listing 3.2 shows an example result from using the input and output properties described above to execute the `quick-check` function from Clojure's [11] *test.check* PBT library [12]. The output shows that a failing test case was found on test iteration *7* and the sequence that caused the initial failure was `[0.9140625 1.9375 -6 -6 -1 2.25]`. The shrinking process then considered *46* nodes of the generator tree and returned `[-1 0 1]` as the smallest value that still fails the output property (shrinking is described in Subsection 3.3.2).

3.3.1 PBT Input Properties

PBT implementations provide a library of generators that can be composed to define input properties of the SUT. This library includes generators that return scalar values such as integers, natural numbers, floating point numbers, characters, strings, keywords, and symbols. The library also includes compound or container generators such as fixed length tuples, variable length vectors, and associative maps. Compound generators are defined in terms of other scalar or compound generators that they contain.

PBT generators take two parameters: a size value and a pseudo random number (PRN). The size parameter indicates the magnitude of the value that the generator should return. The effect of the size parameter is specific to a given generator definition. For example, the size value of an integer generator typically indicates the maximum absolute value of the integer that is returned. The PRN parameter is used to determine what value is returned for a given size parameter value. Compound generators will call their contained generators using a size value that is equal to or smaller than their own size parameter and use a PRN generated deterministically from their own PRN parameter. A combination of generator, size, and PRN always returns the same value which guarantees deterministic behavior so that test executions are reproducible.

The input and output properties (generators and test oracle) are invoked multiple times with varying size and PRN values until a counterexample is discovered where the output properties are violated. This is normally accomplished by invoking the input generators with small initial size values and increasing the size value until an output property violation is discovered. This counterexample is reported to the tester as the initial failing test case. After a failing test case is discovered most PBT systems apply a shrinking process (Section 3.3.2) to find a smaller test case that continues to violate the output properties.

Input property generator functions can be recursive in nature. An example is a generator that returns arbitrary HTML data or any generic tree-like structure. The PBT system is responsible for ensuring that recursion must eventually terminate. This is often accomplished by using the size parameter to determine the maximum depth to which the generator may recurse.

3.3.2 PBT Test Shrinking

PBT is effective at finding bugs in a SUT. However the size values used for the generators may be large when a failing test case is discovered. This large size may be essential to trigger the failure in the SUT or the failure may be detected due to the high coverage of the individual test case. In this case most of the content of the test case is noise that is not related to the output property violation. A noisy test case indicates that a bug exists within the SUT without giving useful indication where in the system the bug exists. PBT systems usually provide a test shrinking capability to address this issue. The capability to shrink tests cases means that the PBT test model can be effective both at discovering failing test cases and providing small versions of those test cases that are useful for identifying the root cause that triggered the failure [23].

Each test case has an in-memory tree representation of the generator nodes with the size and PRN parameters that were used to generate the test case. The shrinking process picks a generator node from the tree and checks whether a smaller version of that generator results in a test case that continues to violate the output property. If so then the new test case is considered smaller than the original one. This shrinking process is repeated until some sort of termination condition is met; either the shrinking process determines that it has reached a local minimum of test case size or it reaches some arbitrary maximum number of search iterations.

One method of shrinking a generator node is to reduce the size value that is used to invoke the generator. However, generator functions are typically defined with custom shrinking processes that accomplish more effective shrinking of the generator data. For example, a bifurcation process can be used to shrink sequences of generated data and a binary search can be used to shrink scalar values [25] [26].

3.3.3 Limitations of PBT

PBT is a powerful automated testing approach that can reveal bugs that are difficult to find with manual example-based tests [23], but there are limitations affecting PBT's utility in important problem domains. First, PBT input property generators are defined with a domain-specific language that is often a subset of or closely related to the implementation programming language of the SUT [27]. This means that the test specifications are difficult for non-developers to create [28] and the specifications are not reusable for testing similar functionality of systems implemented in other languages.

Second, the domain of the input properties is effectively unlimited so choices must be made during test specification design about the distribution of input data. For example, a function that takes five unconstrained 64-bit integers has an input domain that exceeds the number of atoms in the observable universe [29]. When a tester wants to use the PBT model to test a new component in the SUT or to adjust the coverage for an existing test this implies adjusting or creating new test specifications.

Third, the PBT process tends to rediscover the same problems reducing the likelihood of finding bugs triggered only by larger or more complex test cases. This problem is exacerbated by the PBT shrinking process because it does not know if a smaller bug discovered during the shrinking process is actually the same underlying bug or if it is different one [30].

3.4 Instacheck

To address the limitations with PBT we propose a new technique, Instacheck, which builds on the PBT model and addresses the limitations. To validate the Instacheck model we created an implementation of Instacheck in Clojure [11] that leverages the *test.check* PBT library [12] which is based on the QuickCheck model [10]. In this paper we use the term "Instacheck" to refer to both the technique and its Clojure implementation [1]. As with QuickCheck, the Instacheck model can be applied to other languages and testing contexts [24].

3.4.1 Instacheck Grammars

The first PBT limitation that Instacheck addresses is that the test specifications are typically defined using a subset of the language of the SUT. There are at least two reasons for defining test specifications in a more general and abstract language than that of the SUT. The first is to enable reuse of test specifications across SUTs implemented in different programming languages. The second is that the inputs for certain types of systems are already described in an abstract grammar which can readily be translated into the abstract test specification language. One of the motivations for Instacheck is to enable automated testing of web browser rendering engines for which test cases are web pages containing HTML and CSS. The HTML and CSS formats are web standards that are specified using formal grammars.

[1]We selected Clojure due to author familiarity and due to the availability of the *test.check* library for PBT and the *Instaparse* library for defining EBNF parsers.

Section 3.5.6 describes the results of using Instacheck to test web browser rendering engines.

Instacheck currently defines test specifications using a class of formal language grammars known as Extended Backus-Naur form (EBNF). Instacheck leverages the *Instaparse* [14] library for grammar parsing and therefore supports the Instaparse EBNF variant. The supported EBNF syntactical elements, their meaning, and their textual (key) and visual (icon) representation are listed in Table 3.1. The weight column indicates the elements that can be weighted (described in Section 3.4.2).

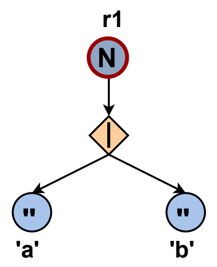
EBNF Syntax	Meaning	Key	Icon	Weight
`A`	non-terminal (LHS)		N	
`A`	non-terminal (RHS)		N	
`"abc"` *or* `'abc'`	literal terminal		"	
`#"abc"` *or* `#'abc'`	regexp terminal		R	
`""` *or* `''` *or* ϵ	epsilon terminal		ε	
`A B` *or* `A, B`	concatenation	:cat	C	
`A | B`	alternation	:alt	|	✓
`A?` *or* `[A]`	optional	:opt	?	✓
`A*` *or* `{A}`	zero or more	:star	*	✓
`A+`	one or more	:plus	+	
`(A B)`	grouping			

Table 3.1: EBNF Syntax

Instacheck Grammar Examples

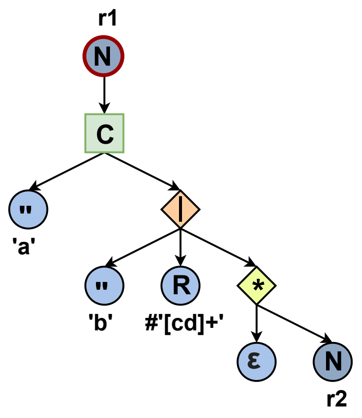
Figure 3.1 shows the directed tree representation of a single EBNF rule `r1 = 'a' | 'b' ;`. This rule defines a grammar that parses a document containing a single character that is either “a” or “b”.

Figure 3.2 shows the directed tree representation of a more complicated rule.



```
r1 = 'a' | 'b' ;
```

Figure 3.1: Grammar with one simple rule containing alternation.



```
r1 = 'a' ( 'b' | #'[cd]+' | r2* ) ;
```

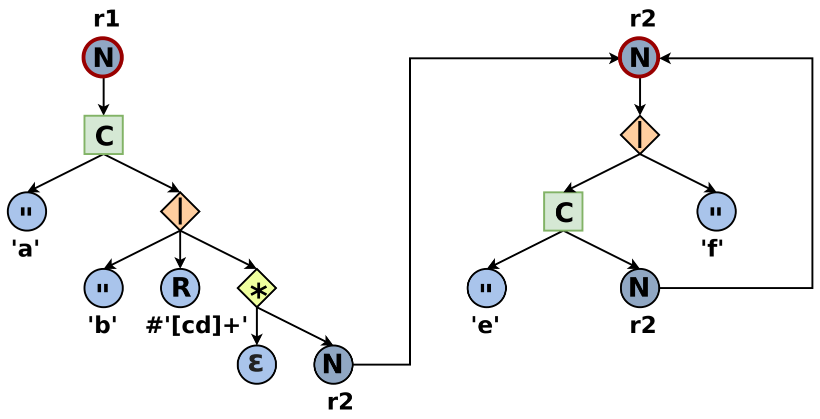
Figure 3.2: Grammar with one rule containing a concatenation, an alternation, a 0 or more repetition (star), and all the terminal types supported by the syntax.

The grammar in Figure 3.2 is incomplete because it has a leaf node that refers to a non-terminal `r2` on the right side of the rule that is not defined. Figure 3.3 expands the grammar to include the definition of the `r2` rule.

Each separate production rule in an EBNF grammar is a directed acyclic graph (DAG). The grammar as a whole may contain cycles where non-terminals in leaf position of a production rule refer to another rule definition including a reference back to the root of the current rule (self-recursive) or to a "parent" of the current rule (mutually recursive).

3.4.2 Grammar Weights

Instacheck uses a system of weights that can be modified to adjust the generated test cases. This is used to address the other PBT limitations. An Instacheck grammar contains weighted nodes that have multiple child paths that may be selected during test case generation. The



```
r1 = 'a' ( 'b' | #'[cd]+' | r2* ) ;
r2 = 'e' r2 | 'f' ;
```

Figure 3.3: Complete grammar with two rules.

weight values affect the probability distributions of traversed paths during value generation. Instacheck provides default uniform weight values for the generators but the weight values can be manually configured prior to a test run or automatically adjusted across multiple test runs. This capability enables Instacheck to address the PBT limitations related to coverage.

The original Haskell QuickCheck implementation can be applied to individual routines of the SUT by performing a naive translation of the type definitions of those routines to generator functions. For simple routines with reasonably constrained input types this can provide a sufficient level of code coverage. However, for more complex routines and when QuickCheck is used at a higher level (modules or whole programs) it is usually necessary to write custom generators to generate test cases with the desired code coverage because generators derived only from the type definitions are unlikely to activate all code paths within the SUT. Grammar weights allow directed code coverage without requiring new custom generators to be created manually.

3.4.3 Grammar Paths, and Treks

The nodes and edges within a grammar graph are identified by *Instacheck paths*. An Instacheck path is a tuple containing a rule keyword followed by zero or more alternating node type keywords and child edge identifiers. In Figure 3.1 the path of the non-terminal root node of the rule is `[:r1]`. The path of the *:alt* node (◇) in the middle of the graph is `[:r1 :alt]`. The paths to the terminal strings "a" and "b" (") are `[:r1 :alt 0]` and `[:r1 :alt 1]` respectively.

The *:alt* (◇) node type uses zero-based indexes to identify child edges within grammar paths. The *:opt* (?) and *:star* (*) node types use `nil` and `0` to identify child edges within grammar paths. When the grammar is used in a parsing context, a `nil` child edge means no text was matched by a child tree and `0` means either one match or one or more matches by the child tree for *:opt* and *:star* node types respectively. When the grammar is used in a generator context a `nil` child edge means no text is produced and `0` means the child tree produces either once or one or more times for *:opt* and *:star* node types respectively.

Instacheck introduces a class of data structures called *treks* to represent grammar graph information. Each trek is a shallow (single level) associative map of grammar paths to values of some type. There are multiple types of treks used in Instacheck but we will focus on *value treks* and *wtreks* (or *weight treks*).

A *value trek* maps grammar paths to leaf node values. Each path in a value trek represents a possible traversal of the grammar from a root non-terminal (N) to leaf terminals / non-terminal (N, ", R, or ε). A value trek is an enumeration of all Instacheck grammar leaf nodes indexed by the paths to those nodes from the root of the grammar. Value treks are a flattened view of the graph that are used for efficient lookup and modification of leaf nodes. Because the paths contain the full node and edge traversal to every leaf node, this means a *value trek* can be transformed to and from a full graph representation.

A *wtrek* (*weight trek*) maps grammar paths to grammar weight values. The weights in a wtrek are used by Instacheck to determine the probability that a child path is chosen during test case generation. Serialized wtreks are used to durably store and to configure grammar weights and serve as a convenient format for end-user customization of the weights. Each path in a wtrek represents a traversal of the grammar from a root non-terminal (Ⓝ) to the child edge of a weighted node (⟨|⟩, ⟨?⟩, or ⟨*⟩). For a given grammar the paths in a value trek and wtrek are partially overlapping sets. A value trek contains paths to unweighted leaf nodes while a wtrek does not. A wtrek contains paths to internal (non-leaf) weighted edges while a value trek does not.

Value Trek Example

Listing 3.3 shows the value trek for the grammar in Figure 3.3. The value trek contains eight path to value mappings. The paths are highlighted in Figure 3.4 with dotted lines. The `:r1` grammar rule has a tree with five paths from root non-terminal node to leaf terminals / non-terminals. The first five key-values of the value trek in listing 3.3 define the `:r1` rule. The `:r2` grammar rule has a tree with three paths from root non-terminal node to leaf terminals / non-terminals. The final three key-values of the value trek in listing 3.3 define the `:r2` rule.

```
{
 [:r1 :cat 0]                        "a",
 [:r1 :cat 1 :alt 0]                 "b",
 [:r1 :cat 1 :alt 1]                 #"[cd]+",
 [:r1 :cat 1 :alt 2 :star nil]       "",
 [:r1 :cat 1 :alt 2 :star 0]         :r2,
 [:r2 :alt 0 :cat 0]                 "e",
 [:r2 :alt 0 :cat 1]                 :r2,
 [:r2 :alt 1]                        "f"
}
```

Listing 3.3: Value trek for the Figure 3.3 grammar.

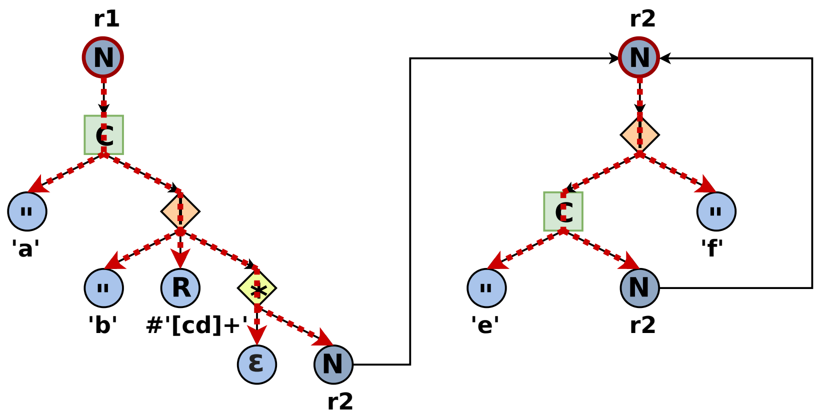


Figure 3.4: Value trek paths shown visually with red dotted lines for the Figure 3.3 grammar.

Weight trek (wtrek) Example

Listing 3.4 shows the wtrek representation of the grammar in Figure 3.3. The wtrek contains seven path to weight mappings. The paths are shown visually in Figure 3.5 with red dotted lines. The `:r1` grammar rule has a tree with five paths from root non-terminal node to weighted node child edges. The first five key-values of the wtrek in listing 3.4 define the weights for the `:r1` rule. The `:r2` grammar rule has a tree with two paths from root non-terminal node to leaf terminals / non-terminals. The final two key-values of the value trek in listing 3.4 define the weights for the `:r2` rule. The `[:r2 :alt]` node has two children with weights of 200 and 100 respectively. This means that during test case generation, the probability that the first child path (`[:r2 :alt 0]`) will be chosen is twice as high as the second child path (`[:r2 :alt 1]`). Paths with higher weights are also considered "simpler" by the Instacheck test case shrinking process.

In Figure 3.5 the *:star* node (0 or more) has an *epsilon* child indicated by the `nil` as the final path element. The separation of the *epsilon* (empty) child of *:star* and *:opt* nodes gives more control over the test cases generated by the grammar.

```
{
 [:r1 :cat 1 :alt 0]             100,
 [:r1 :cat 1 :alt 1]              50,
 [:r1 :cat 1 :alt 2]              50,
 [:r1 :cat 1 :alt 2 :star nil]    10,
 [:r1 :cat 1 :alt 2 :star 0]      90,
 [:r2 :alt 0]                    200,
 [:r2 :alt 1]                    100
}
```

Listing 3.4: Wtrek (weight trek) for the Figure 3.3 grammar.

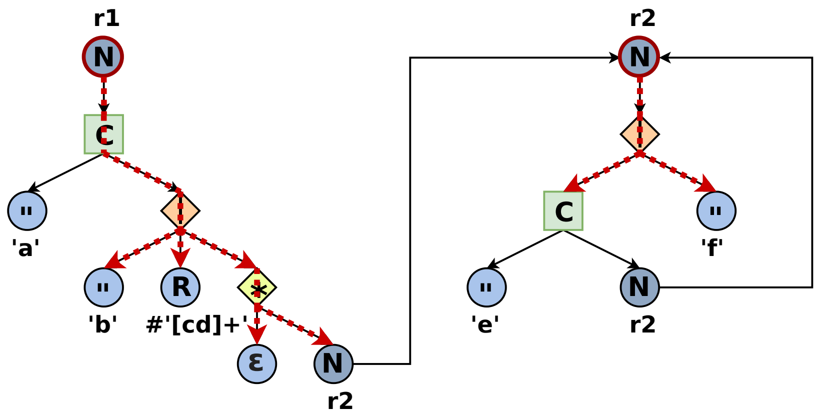


Figure 3.5: Wtrek (weight trek) paths shown visually with red dotted lines for the Figure 3.3 grammar.

When the weight value for a child path is zero, then the probability of that path being chosen during generation is also zero. This means that the child node is *unreachable* from the parent node and also from the grammar as a whole. A node is *reachable* if there is a path from the root of the grammar to the node and all weighted edges that are crossed have a weight greater than zero.

3.4.4 Grammar Weight Reduction and Propagation

Instacheck defines an automatic weight modification approach that can increase multiple run coverage by reducing the probability of traversing the grammar paths that were already used to generate counterexamples. This is achieved by modifying the active wtrek configuration that Instacheck will use for subsequent test runs. Automatic weight reduction addresses the third limitation with PBT where the same counterexamples are found repeatedly and may hide more complex counterexamples. There are three components of Instacheck weight reduction: selecting weights to reduce, reducing the selected weights, and propagating weight reductions to prevent invalid weight configurations. An invalid weight configuration is one in which a weighted node is reachable from the root node but has no reachable children because all of its child edges have a zero weight.

The weight selection algorithms currently supported by Instacheck reduce the active wtrek using a parsed frequency trek of each path that is traversed while parsing a test case. This path frequency trek is a form of wtrek where the weights represent the frequency with which the respective paths were found in the parsed test case. In a multiple test run scenario the test case that is parsed is the final shrunk test case from the most recent test run. Instacheck currently has three selection algorithms:

1. *:weight* - Random choice weighted by the weight value of paths in the parsed path frequency trek.

2. *:dist* - Random choice weighted by the distance of the weighted path from the root of the grammar using Djikstra's Shortest Path First algorithm [31].

3. *:weight-dist* - Random choice weighted by a combination of *:dist* and *:weight*.

Once a path is selected for reduction, a reducer function is then applied to modify the weight for that path in the active wtrek. The reducer function takes the current wtrek weight and the weight from the parsed path frequency trek and returns a new weight that is used to update the active wtrek. The reducer function must eventually reduce the weight to 0.

Once a wtrek weight value is reduced, the propagation algorithm is applied in order to prevent invalid weight configurations. Instacheck currently has three propagation algorithms:

1. *:zero* - If all siblings of a node are zero, reduce nearest parent edge weights to zero.

2. *:max-child* - If all siblings of a node have a weight that is less than parent edge weight then reduce that parent edge weight to the largest sibling weight.

3. *:reducer* - If all siblings of a node are zero, reduce parent edge weights by reducer-fn function and distribute the removed weights to valid (no removed descendant) child edges of node.

Listing 3.5, 3.6, and 3.7 show pseudocode for the three propagation algorithms.

We found that these algorithms for selection, reduction, and propagation have a reasonable balance between increasing coverage and algorithmic efficiency (see Section 3.5.3). Particular characteristics of the underlying grammars being tested may have a significant impact on the effectiveness of selection, reduction, and propagation strategies.

```
pend := selected-and-reduced-nodes
while pend:
  node   := pop(pend)
  mcw    := max-child-weight(node)
  if mcw > 0: continue at while
  foreach pnode of parents(node):
    push(pend, pnode)
    wtrek[pnode] := mcw
```

Listing 3.5: Weight Propagation Algorithm *:zero* Pseudocode

```
pend := selected-and-reduced-nodes
while pend:
  node   := pop(pend)
  mcw    := max-child-weight(node)
  foreach pnode of parents(node):
    if pnode child weight towards node > mcw
      then:
        push(pend, pnode)
        wtrek[pnode] := mcw
```

Listing 3.6: Weight Propagation Algorithm *:max-child* Pseudocode

```
pend := selected-and-reduced-nodes
while pend:
  node   := pop(pend)
  mcw    := max-child-weight(node)
  if mcw > 0: continue at while
  acc    := 0
  foreach pnode of parents(node):
    tmp := wtrek[pnode]
    wtrek[pnode] := reducer-fn(wtrek[pnode])
    acc   += tmp - wtrek[pnode]
    if max-child-weight(pnode) == 0:
      push(pend, pnode)
  cnodes := children-with-valid-descendants(node)
  foreach cnode of cnodes:
    wtrek[code] += acc / count(cnodes)
```

Listing 3.7: Weight Propagation Algorithm *:reducer* Pseudocode

3.4.5 Reproducing and Shrinking Existing Test Cases

The process described in Section 3.4.4 uses a path frequency trek parsed from a generated test case and uses those paths to reduce the weights in the active wtrek. The grammar paths with reduced weights will be less likely to be traversed during test case generation, which reduces the likelihood of generating similar test cases.

Using an inverse weight adjustment process, we can parse a path frequency trek from an existing test case that was not generated by Instacheck and use the parsed frequencies to increase rather than decrease the probability of those grammar paths being traversed. This will increase the probability that Instacheck will produce similar test cases to the parsed external test case. The weights in the path frequency trek capture the weighted node path probability distributions of the parsed test case and these weights can be used directly as the active wtrek configuration. New test cases generated with this weight configuration may not be identical to the original test case but they will have a similar frequency distribution of paths taken through the grammar during generation. This increases the likelihood that the bug that triggered the counterexample in the external test case will be detected in new generated test cases (see Section 3.5.5).

The normal PBT shrinking process cannot be applied directly to external test counterexamples because there is no in-memory generator tree available on which to apply the process. The use of weight configuration to find similar counterexamples allows the shrinking process to be applied because the generator tree is available for test cases generated by the system itself.

3.5 Results

3.5.1 Test Grammars

To validate the Instacheck model we use a two variant EBNF grammar that describes simple web page-like test cases containing tag elements, tag attributes, properties and property values. These two variants are referred to as MG4 (markup grammar degree 4) and MG8 (markup grammar degree 8) respectively. The EBNF for the first variant is shown in listing 3.8. The second grammar is similar but has eight alternations for each tag, attribute, property, and property value rather than four. Listing 3.9 shows a simple test case generated by the MG4 grammar.

The features and failures in test cases generated by MG4 or MG8 grammars are quantified using a TAPV. The test case in listing 3.9 has a single TAPV [*tag1*, *attr3*, *pname0*, *pval2*] which is abbreviated as *1302*. Listing 3.10 shows a larger test case generated by grammar MG4. This test case contains three TAPVs: *2223*, *2230*, *1201*. Each generated tag may contain more than one TAPV because a tag may contain multiple attributes, an attribute may contain multiple properties, and a property may have multiple property values. TAPVs do not span multiple tags even if a tag element is contained within another tag element (recursion).

Each TAPV defines a "feature unit" of the theoretical system that renders these test cases (the system under test). When three or four of the TAPV numeric indexes match, then the feature has incorrect behavior. In other words, TAPV *1011* triggers a misbehavior (counterexample) in the underlying SUT because the tag, property and property value indexes match. The test case in listing 3.10 contains three unique TAPVs of which TAPV *2223* triggers a failure (counterexample). The test case in listing 3.11 has seven unique TAPVs. TAPV *2001* occurs twice but only counts as a single TAPV for feature coverage purposes. This test case also contains a counterexample TAPV *2000*.

The number of possible TAPVs for a grammar with degree d is given by d^4 and the number of failing TAPVs is given by $4d(d-1)+d$. The MG4 and MG8 grammars have a total of 256 and 4096 possible TAPVs and 52 and 232 failing TAPVs respectively.

```
test = elem+
elem = '<tag0' (' ' attr)+ '>' elem* '</tag0>'
     | '<tag1' (' ' attr)+ '>' elem* '</tag1>'
     | '<tag2' (' ' attr)+ '>' elem* '</tag2>'
     | '<tag3' (' ' attr)+ '>' elem* '</tag3>'
     | '<tag4' (' ' attr)+ '>' elem* '</tag4>'
attr = aname '="' prop (' ' prop)* '"'
prop = pname ':' (' ' pval)+ ';'

aname  = 'attr0'  | 'attr1'
       | 'attr2'  | 'attr3'
pname  = 'pname0' | 'pname1'
       | 'pname2' | 'pname3'
pval   = 'pval0'  | 'pval1'
       | 'pval2'  | 'pval3'
```

Listing 3.8: Markup grammar degree 4 (MG4)

```
<tag1 attr3="pname0: pval2;">
</tag1>
```

Listing 3.9: Test case generated by MG4 containing a single unique TAPV: *1302*.

```
<tag2 attr2="pname2: pval3;
             pname3: pval0;">
  <tag1 attr2="pname0: pval1;">
  </tag1>
</tag2>
```

Listing 3.10: Test case generated by MG4 containing three unique TAPV: *2223*, *2230*, *1201*.

```
<tag2 attr3="pname1: pval0 pval3;
             pname3: pval0;">
</tag2>
<tag2 attr0="pname0: pval0 pval1 pval1;
             pname3: pval2;
             pname1: pval0;">
</tag2>
```

Listing 3.11: Test case generated by MG4 containing seven unique TAPVs: *2310*, *2313*, *2330*, *2000*, *2001*, *2032*, *2010*.

3.5.2 Controlling Coverage

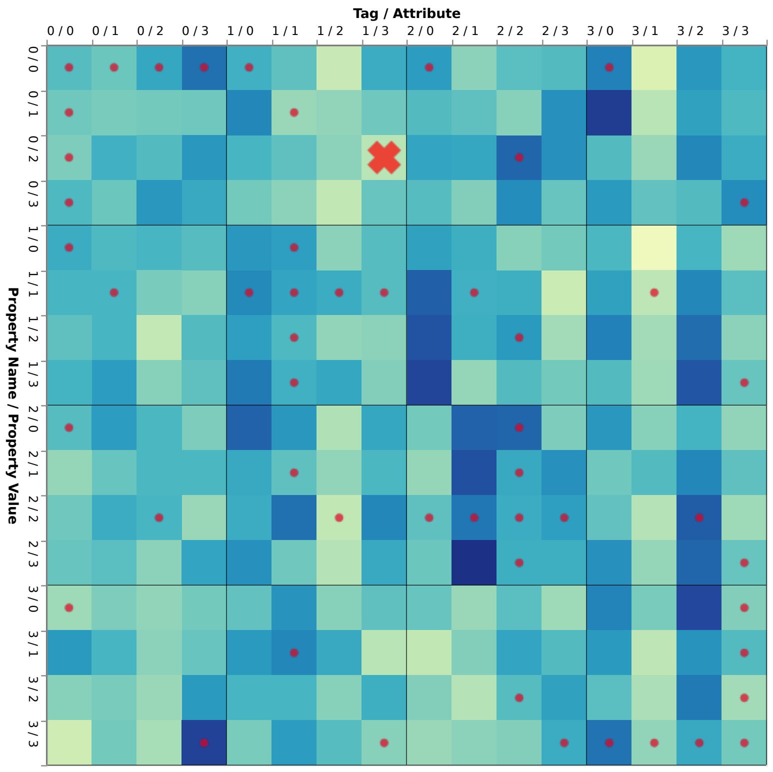
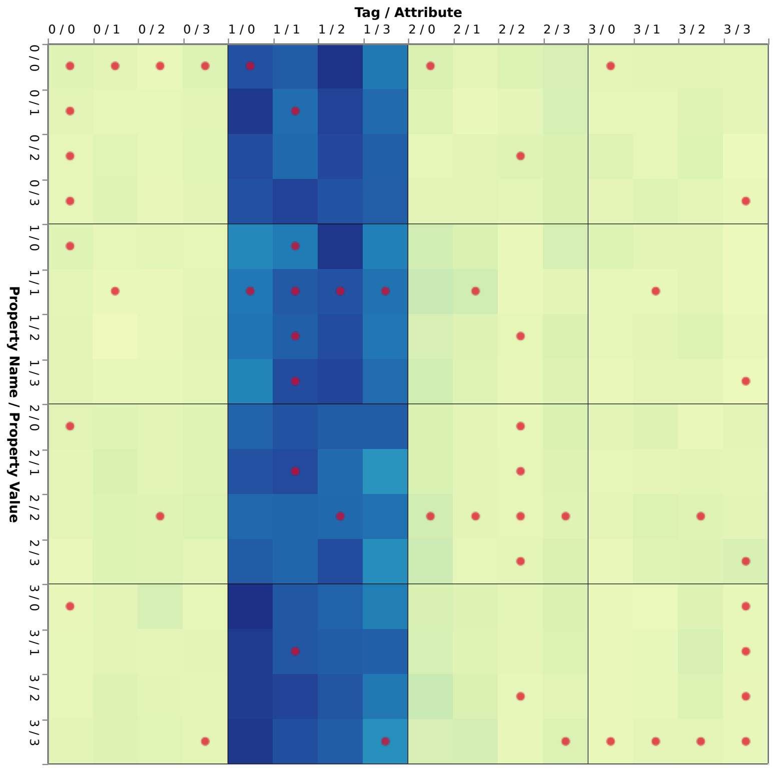


Figure 3.6: Coverage for the MG4 grammar with default grammar weights. The X axis delimits tags (large boxes) and attributes (small boxes) and the Y axis delimits properties (large boxes) and property values (small boxes).



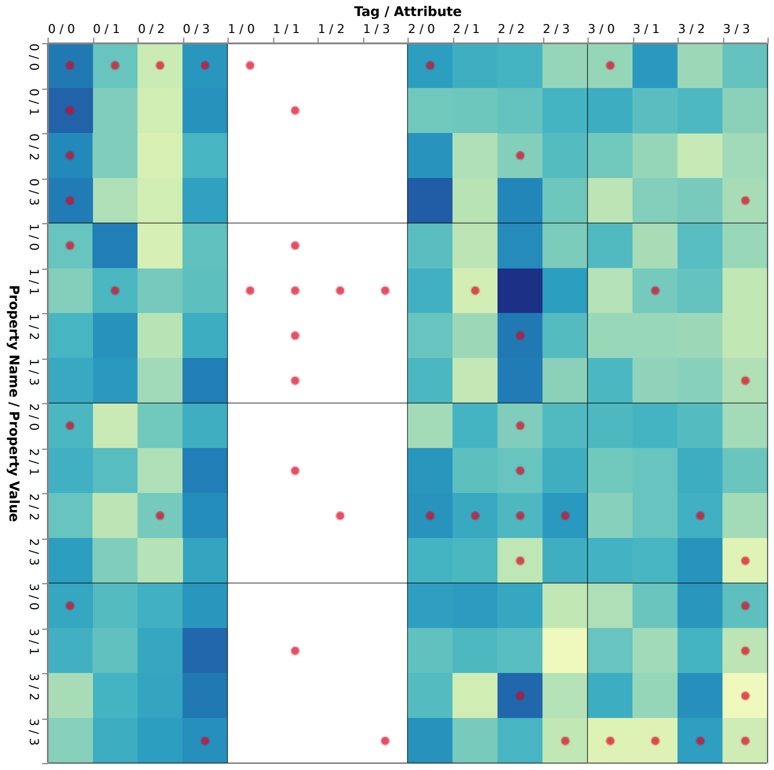
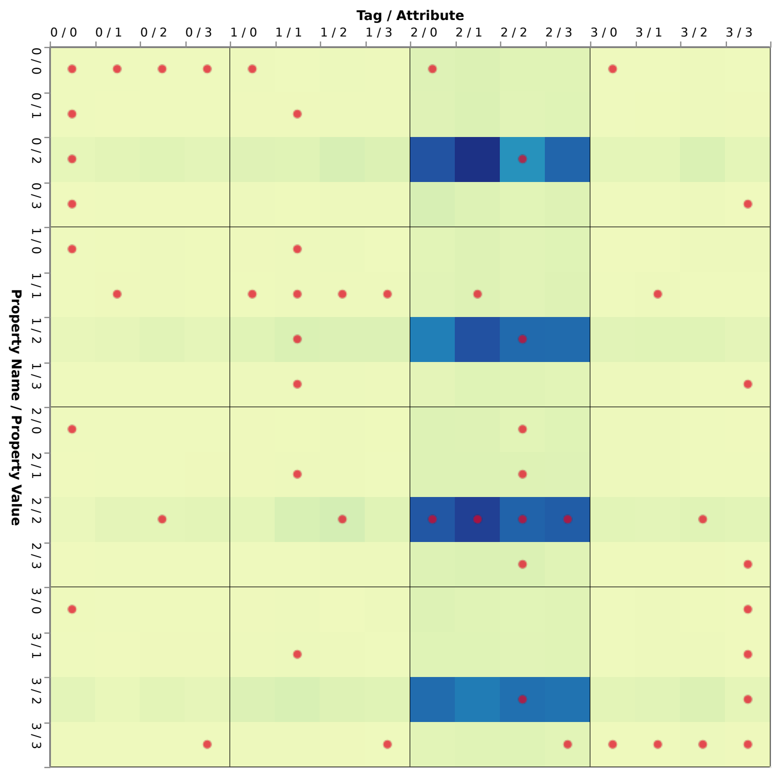


Figure 3.7: Coverage for the MG4 grammar one weight adjusted. First figure shows coverage with the weight of *tag1* multiplied by ten. Second figure shows coverage with weight of *tag1* set to zero.



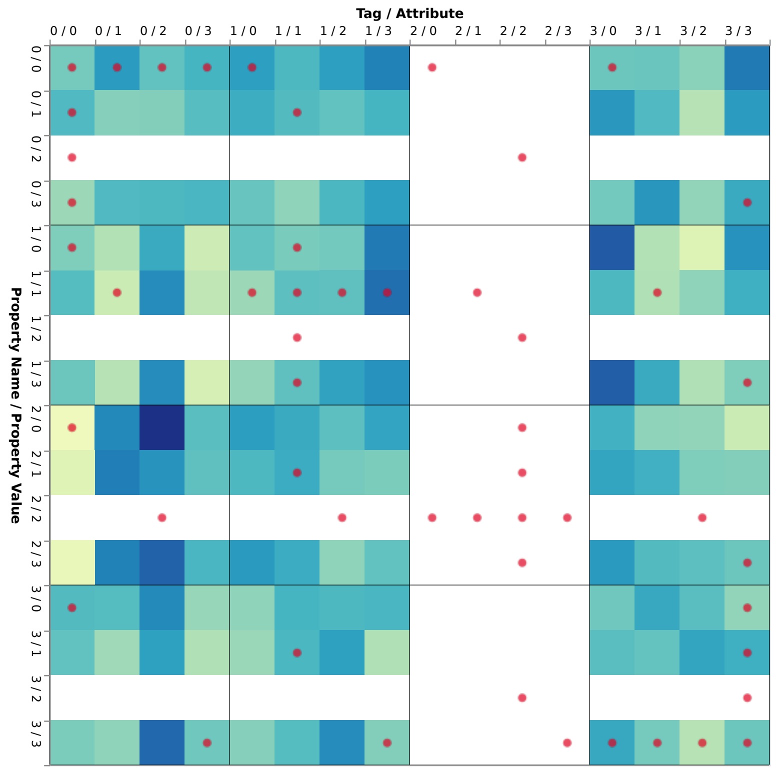


Figure 3.8: Coverage for MG4 grammar with two weights adjusted. Left figure show weights for *tag2* and *prop2* multiplied by ten. Right figure shows weights for *tag2* and *prop2* set to 0.

Figures 3.6 - 3.8 show heatmap representations of TAPV coverage that result from generating random test case samples with the MG4 grammar. Each small box represents a single TAPV and the darker the box the greater the frequency of that TAPV occurring within the generated samples (white means there was no coverage of that TAPV). The TAPVs are arranged with tag indexes and attribute indexes along the horizontal axis and properties and property values along the vertical axis. The large boxes delineate tags and properties that contain attributes and property values respectively. The dots mark the TAPV squares that trigger failures.

Figure 3.6 shows the result of using the MG4 grammar with uniform weights to generate 100 random samples with a maximum generation size of 6. The "X" in the figure shows the location of TAPV *tag1*, *attr3*, *pname0*, *pval2* (*1302*). This is the TAPV from the test case in listing 3.9.

Figure 3.7 shows two MG4 coverage heatmaps that result from repeating the same sampling process but with adjusted weights for *tag1* (grammar path `[:elem :alt 1]`). The first heatmap shows the result of multiplying the weight by ten. The second heatmap shows the result of setting the weight to zero. The left heatmap shows that the coverage of all TAPVs with *tag1* increases while other TAPV coverage decreases. The right heatmap shows that the coverage of TAPVs with *tag1* drops to zero while other TAPV coverage increases. Figure 3.8 shows two MG4 coverage heatmaps that result from repeating the same sampling process but with weights adjusted up and down for both *tag2* (grammar path `[:elem :alt 2]`) and *pval2* (grammar path `[:pval :alt 2]`).

This shows that manual grammar weight adjustment can be used to target specific features of the SUT either by increasing coverage for features of interest or by decreasing or eliminating coverage for uninteresting features.

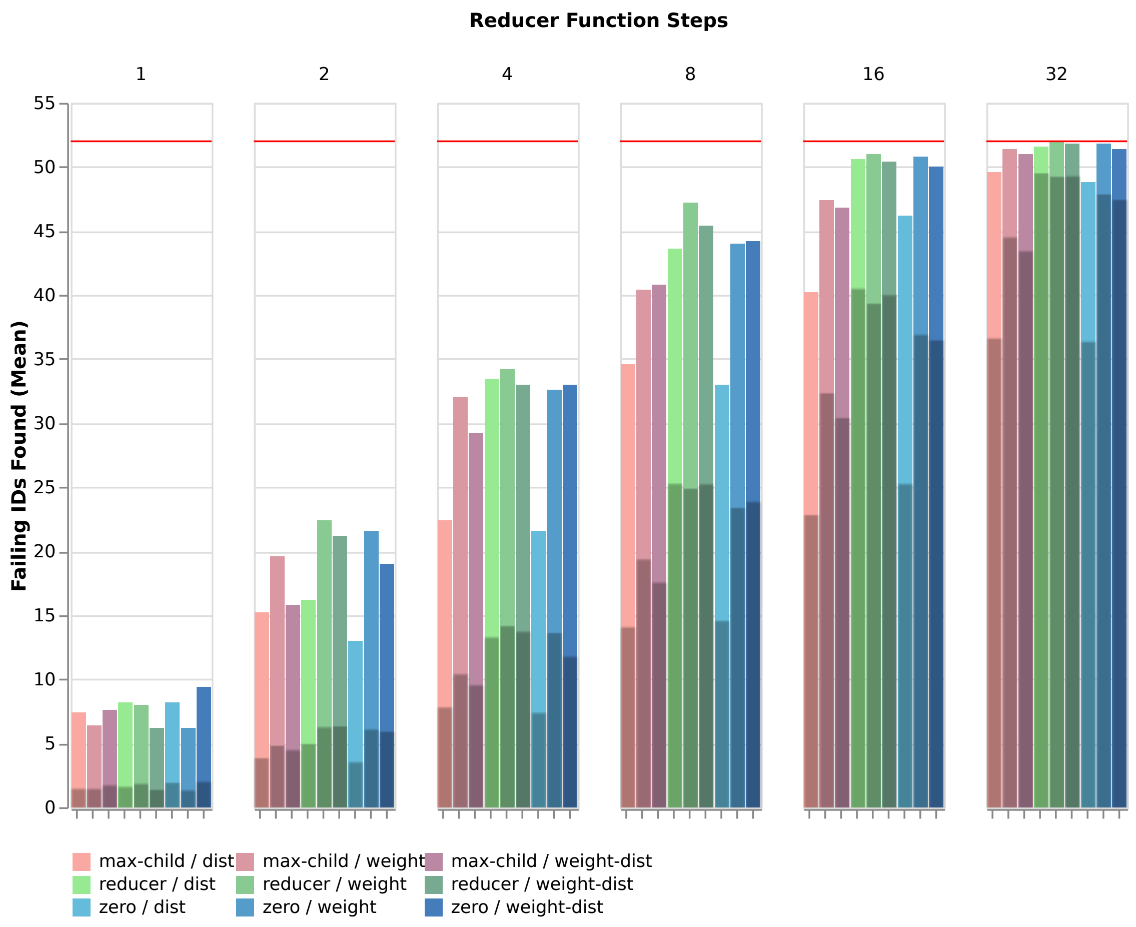


Figure 3.9: Failing TAPVs found for each selection and propagation algorithm across 6 different reducer function steps using the MG4 grammar. Counts are averaged across 5 sample runs.

3.5.3 Increasing Coverage with Weight Reduction

Figures 3.9 and 3.10 show the results of using different combinations of Instacheck reduction algorithms over multiple test runs. Each primary bar represents a combination of propagation algorithm (*:dist*, *:weight*, *:weight-dist*) and selection algorithm (*:max-child*, *:reducer*, *:zero*). The bars are grouped by the number of reducer function steps that were used. The reducer step count is the number of times that the weight is reduced before it reaches zero and can no longer be reduced (starting from 100). For example, a reducer function that divides the weight by four would have four steps, because it results in the following integer progression (when starting from 100): 25, 6, 1, 0.

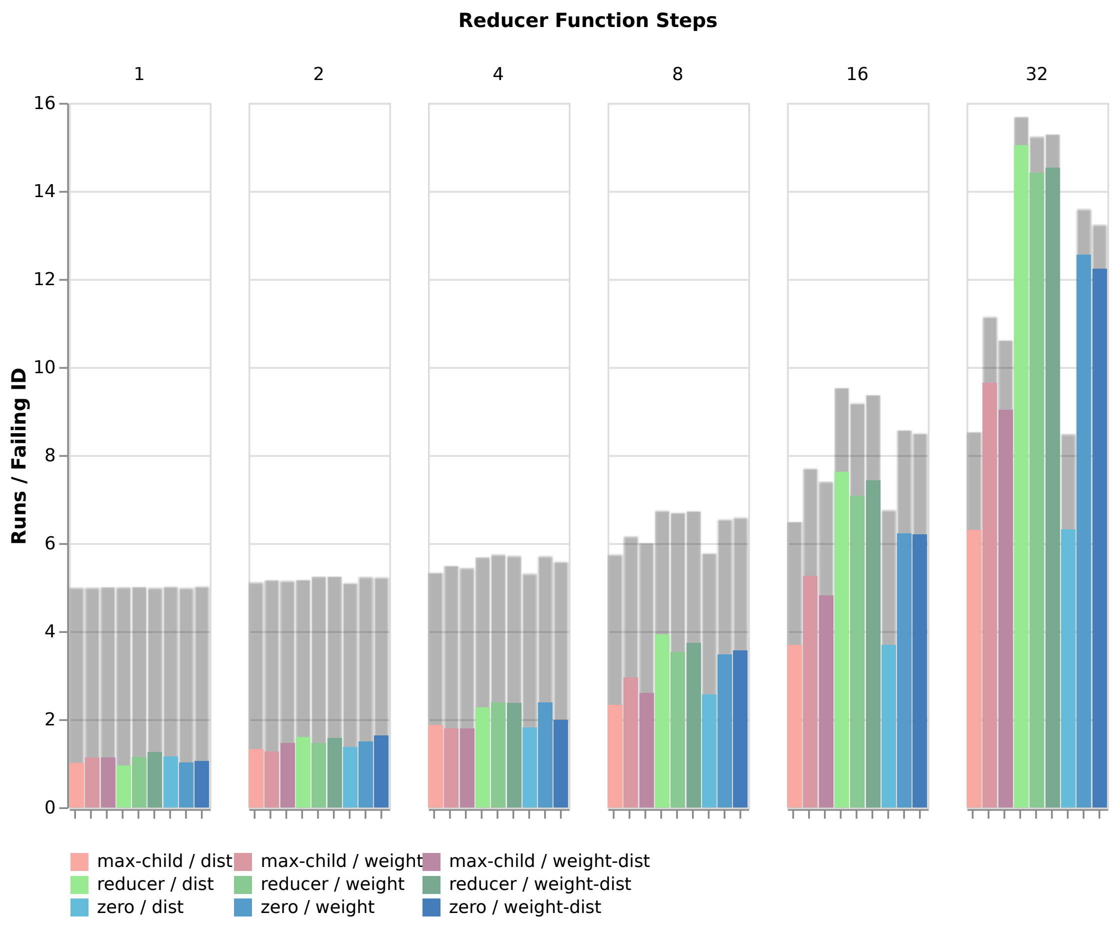


Figure 3.10: Number of test runs required to find each failing TAPVs for each selection and propagation algorithm across 6 different reducer function steps using the MG4 grammar. Counts are average across 5 sample runs.

Each algorithm combination is tested by starting with a uniform active wtrek configuration. After each run, the active wtrek is adjusted by the algorithms being tested and the result is used for the subsequent run. This process continues until the active wtrek can no longer be reduced, because the active wtrek configuration has reached an invalid state (propagation has reached the grammar root node). This process is repeated 5 times for each algorithm combination and the results are averaged.

Figure 3.9 shows the average number of failing TAPVs discovered at the conclusion of the test for each algorithm combination. The horizontal line at 52 indicates the number of failing TAPVs that exist in the MG4 grammar. The darker overlay bars represent the expected number of failing TAPVs that would be discovered if the test was for the same number of runs with no reduction being applied. The expected coverage value M with n elements and x samples with replacement is:

$$E[M] = n\left(1 - \left(\frac{n-1}{n}\right)^{x}\right)$$

The use of reduction algorithms results in an increase in the number of failing TAPVs discovered for a given number of test runs for every algorithm combination. The propagation / selection combination of *:reducer / :weight* finds the most failing TAPVs before no further reductions can be applied. However, it is also important to consider efficiency. Figure 3.10 shows the average number of runs that is required to find each failing TAPV. Smaller bars indicate greater efficiency. The light bars show the expected number of runs per failing TAPV if the test was run for the same number of runs as the corresponding primary bar and with no reductions applied. In all cases, using the reduction algorithms is more efficient than testing with no reductions. The most efficient algorithm combinations are *:max-child / :dist* and *:zero / :dist*. However, these algorithms tend to perform worse in terms of the total number of failing TAPVs discovered when run to completion.

These results show that the use of Instacheck weight reduction algorithms increase coverage across multiple runs and are more efficient at finding failing test cases than repeated test runs without reduction. The choice of reduction algorithm combination depends on the desired balance of discovered failing test cases and the efficiency of finding each test case.

3.5.4 Shrinking

Figure 3.11 shows 10 runs of the `quick-check` process using the MG8 grammar with a maximum generation size parameter set to 6. The X axis shows the run iteration when the initial test failure was discovered (circle) and when the final shrunk test case was reported (square). The difference between the shrunk iteration and the initial failure iteration represents the number of iterations that were used to complete the shrinking process. The Y axis is a logarithmic axis that shows the byte count size of the test case of the initial test case discovery and of the final shrunk test case.

Several of the lines in Figure 3.11 show momentary increases in the size of the test case during shrinking. There are two reasons for this. The first is that this figure is reporting what Clojure's *test.check* reports as the current smallest test case, which represents a minimum in the current branch of exploration and not necessarily the smallest test case globally. A second reason is that during the translation from the abstract generator tree to an actual test case there are cases where the tree may be considered smaller even though it results in more bytes in the translated test case. For example, property name strings use six bytes in a test case while property value strings only use five bytes. However, both of these represent a single node in the generator tree and would be considered equivalent in size by the test system.

In Figure 3.11, the initial failure size has a mean of 1357 bytes and a shrunk size mean of 125 bytes. This represents a 91% decrease from the initial mean to the shrunk mean. This is a small sample using a synthetic grammar; however, this is consistent with the results from browser render testing discussed in Section 3.5.6 where we saw a 94% decrease from the initial mean size (4784 bytes) to the shrunk mean size (308 bytes).

3.5.5 Reproducing Existing Test Cases

Figures 3.12 and 3.13 show the effect of using Instacheck to parse existing test cases in order to reproduce the same failure (counterexample). For this test we consider only a single TAPV to be a failing feature. The X axis shows the number of TAPVs in the test case that were parsed to extract a path frequency trek. In both figures, the left-most part of the graph represents test cases containing 256 TAPV values. This means that the test cases contain 255 TAPVs that are not relevant to the failure condition. Each subsequent tick mark represents a 50% reduction in the number of irrelevant TAPVs (noise) in the test case. The right-most tick represents a test case that contains only the failing TAPV. The Y axis

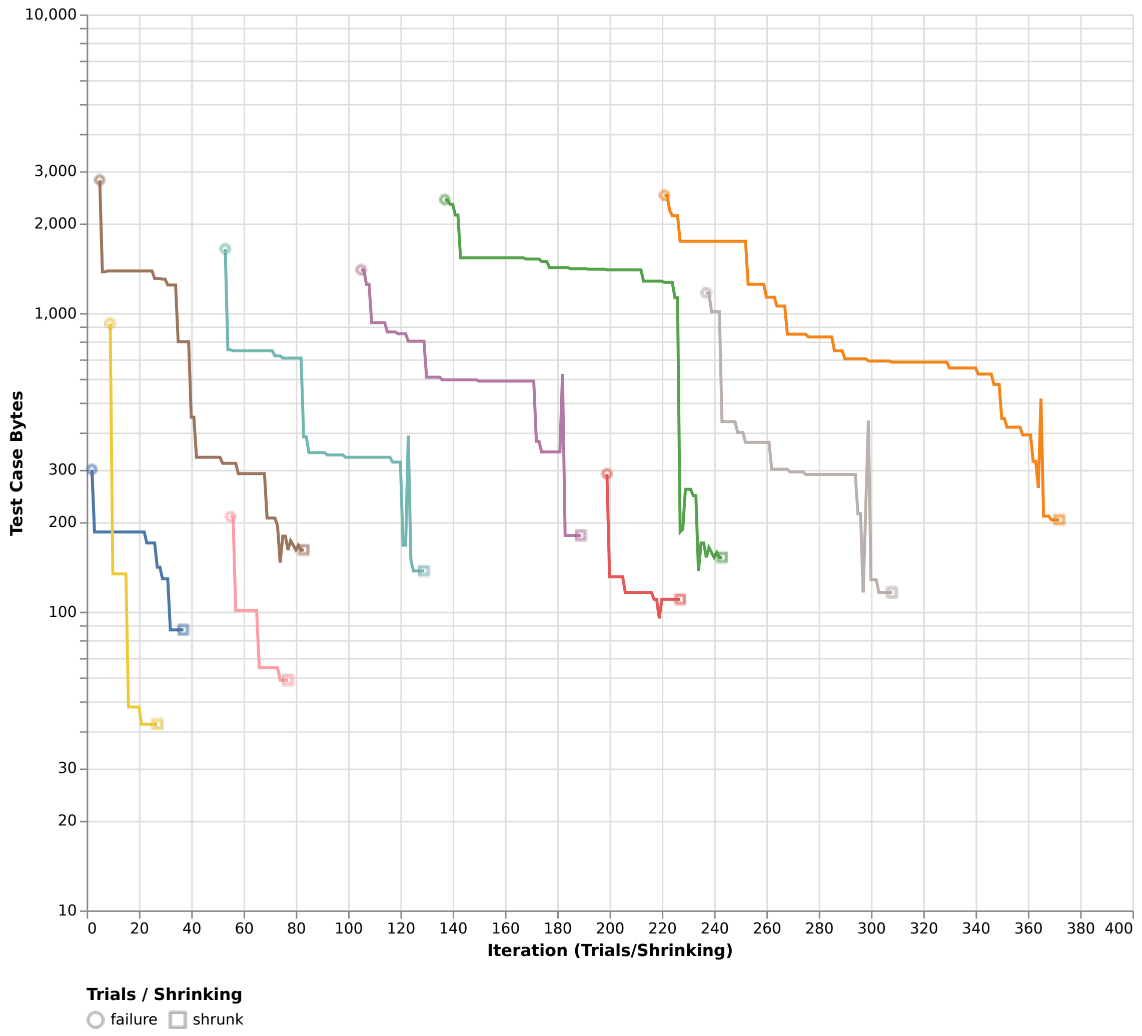


Figure 3.11: Example of shrinking process using MG8 grammar over 10 runs (with max size 6). The iteration and initial failure size are marked with a circle and the iteration and final shrunk size for that run are marked with a square.

represents the number of TAPVs that were checked during a single *quick-check* run before a failing test case was discovered.

For each initial test case size, a random test is generated with the given number of unique TAPVs. One of the TAPVs is randomly chosen to represent the failing feature. The test case is then parsed by Instacheck to extract a path frequency trek. This trek is then used directly as the active wtrek configuration to run *quick-check* (with a max generator size of 25) until a failing case is discovered. The TAPVs from all iterations of the run are summed to give a measure of the computational effort that is necessary to rediscover a failing test case. The actual elapsed time needed to discover a failure is similar in shape to the number of TAPVs explored but it is noisier because it can be affected by conditions of the test system while TAPV count is not. For each size value the test is run 1000 times (9000 runs total per figure). The line shows the median TAPV count and the band shows the third and second quartiles above and below the median respectively.

One question that arises from comparing the two results is: why does the same amount of noise (non-failing TAPVs in the test case) results in different levels of effort between the MG4 and MG8 grammar? The effect results from the weights in the grammar being attached to components of the TAPVs rather than to whole TAPVs. At higher levels of noise, the weights at every alternation point (tag, attribute, property name, and property value) are likely to have a non-zero weight and a possibility of generating data. The higher degree of possible paths at each alternation point in the MG8 grammar means that generated test cases will be more diverse for a given amount of noise.

The data shows that parsing an existing test case can be an effective way of reducing the effort required to reproduce the failure. It is also clear that parsing of narrower test cases (less noise or irrelevant TAPVs) will result in less effort required to rediscover the failure using Instacheck. This data represents the worst-case case scenario where the is no correlation between TAPV elements. In real-world scenarios, the elements would be likely to be correlated which would increase the reproduction efficiency.

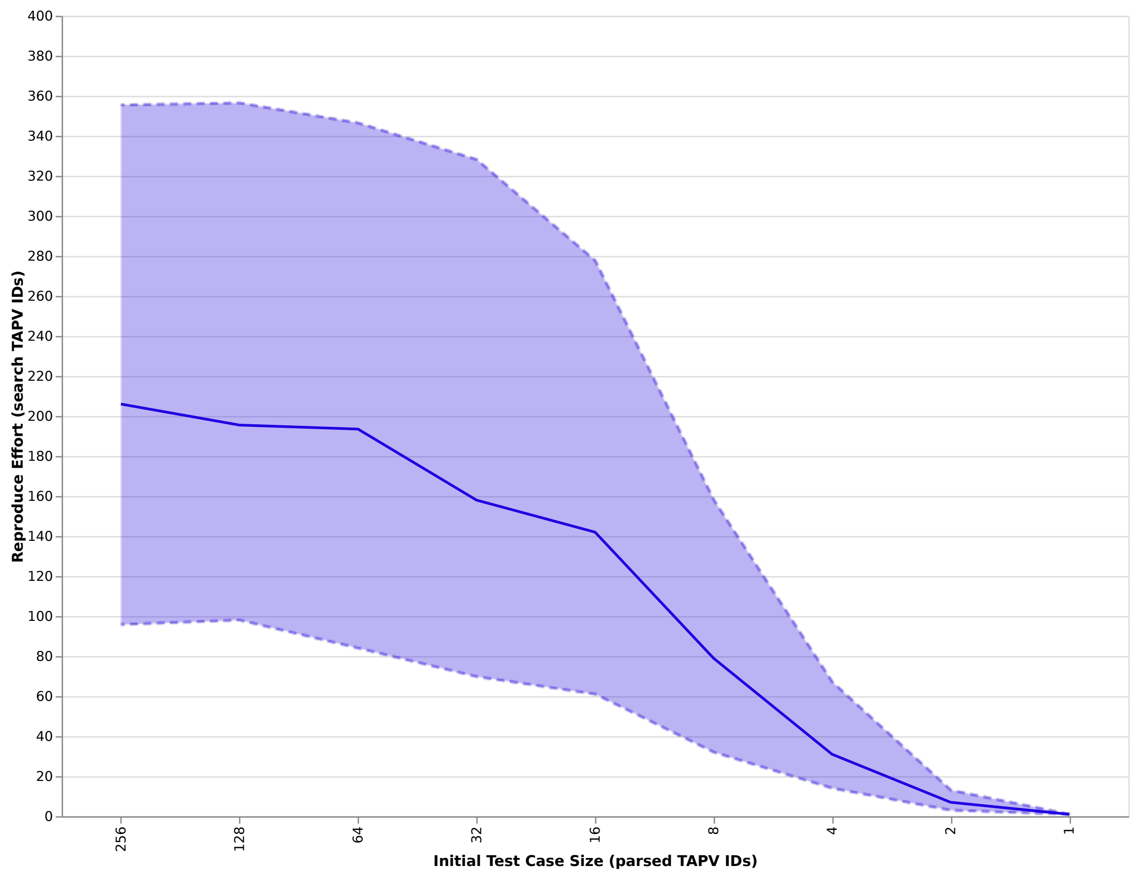


Figure 3.12: Effectiveness of reproducing an existing test case with the MG4 grammar by parsing existing test cases with varying amounts of noise.

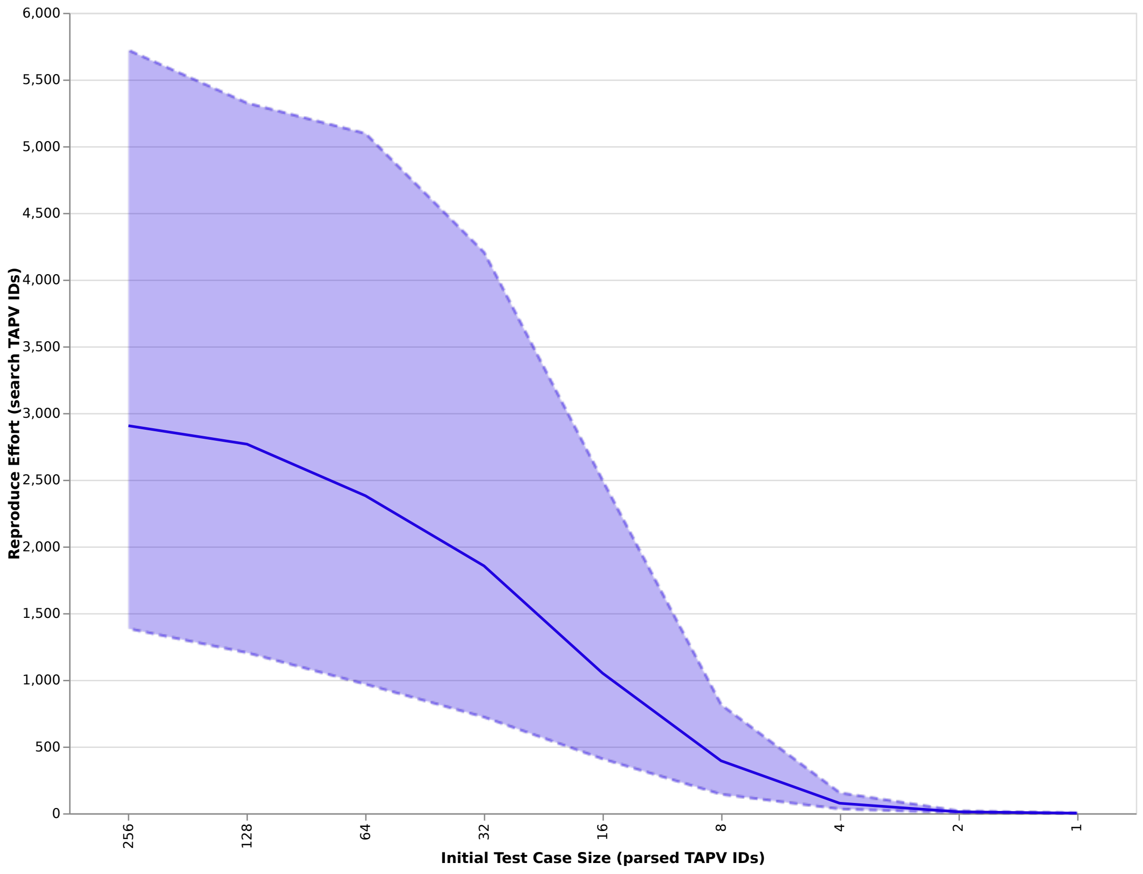


Figure 3.13: Effectiveness of reproducing an existing test case with the MG8 grammar by parsing existing test cases with varying amounts of noise.

3.5.6 Case-study: Detecting Browser Rendering Differences

We validated the Instacheck model on a real-world testing problem by using it to catalog browser web page rendering differences between browser rendering engines. To do this we translated W3C [32] HTML and CSS specification data into EBNF grammars. These grammars were then loaded by Instacheck and used as input properties to generate random but semantically valid web pages. Each generated web page was loaded into a browser consensus pool (differential test). The resulting rendered pages were captured and a disagreement value for each pair of screenshots was calculated as a normalized SSD for each color channel calculated as follows:

$$R(a,b) = \frac{\sum_{x,y} (a(x,y) - b(x,y))^2}{\sqrt{\sum_{x,y} (a(x,y)^2 . \sum_{x,y} b(x,y))^2}}$$

The value calculated for each color channel was averaged and if the value was above a configurable threshold, then the browsers were considered to be in disagreement and the web page test case was considered a failing test case.

Results were generated using Google Chrome 75.0.3770.142, Mozilla Firefox 68.0, and Mozilla Servo (git hash 9451a00). The number of iterations for each run was set to 25 with a maximum generator size parameter of 50. The normalized SSD threshold value was 0.0001. After each test run the weights were reduced using the *:weight* selection algorithm, the *:zero* propagation algorithm, and a 3 step reducer function (weight progression: 100, 10, 1, 0).

Listing 3.12 shows an example of a shrunk test page found during testing where Firefox has different rendering from Chrome and Servo.

```
<body style="hyphens: unset;
             text-emphasis: filled;
             hyphens : unset">
  x
  <div>X</div>
</body>
```

Listing 3.12: Example test case resulting in a rendering difference

The rendered results for this test case are shown for Firefox, Chrome, and Servo in Figure 3.14. Note that font characters are rendered as solid boxes due to use of the Ahem [21] font to reduce normal browser font rendering differences.

We ran two sets of test runs: the first tested Firefox against Chrome and the second compared Servo against Firefox and Chrome. The Firefox/Chrome comparison was executed for 5,000 test runs totaling 89 hours of run time and 712,193 total page load iterations. 1,805 test cases were identified that caused rendering differences of which 1,695 were unique test

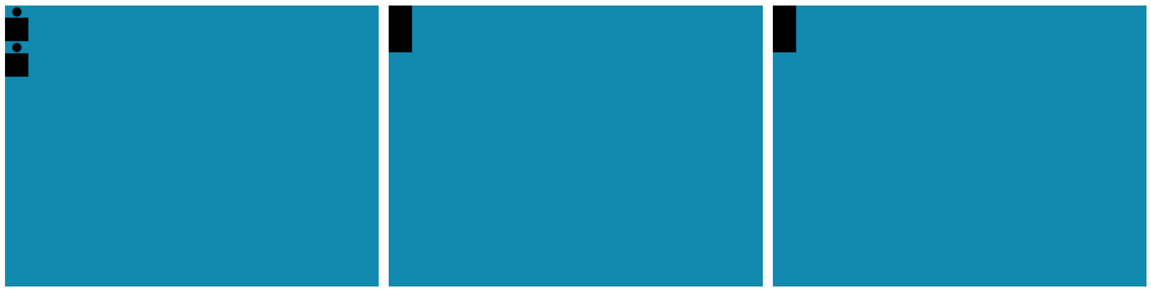


Figure 3.14: Firefox, Chrome, and Servo render for test case in listing 3.12

cases. Across both sets of tests the mean size of the initial test failure was 5823 bytes while the mean shrunk page size was 371 bytes giving a ratio of mean shrink size to mean initial size of 6% (a decrease of 94%).

For the second set of test runs we used Firefox and Chrome as a test oracle to test Servo. We used a consensus algorithm that classified a test case as failing only when Servo's rendering differed and both Firefox and Chrome were in consensus (below the threshold value). To increase testing efficiency we initialized this set of runs by using the reduced weights resulting from the Firefox/Chrome set of test runs. The Servo test was executed for 1000 test runs totaling 60 hours of run time and 112,073 total page load iterations. 257 test cases were identified that caused rendering differences of which all 257 were unique test cases. Finally, we identified 19 confirmed bugs in Servo including a number of rendering related resources leaks that resulted in crashes. Several of the bugs have been resolved and we are continuing to work with Mozilla developers to resolve the remaining bugs.

3.6 Related Work

This section is elided because the content is subsumed by the overall Related Work in chapter 8

3.7 Conclusion

We have presented a new technique called Instacheck that addresses limitations with PBT. We validated the capabilities of Instacheck using a constrained markup language grammar. We were able to successfully apply Instacheck in a browser rendering case-study and identified over 1,600 unique and minimized (shrunk) web page test cases that cause rendering differences between Firefox and Chrome. In addition, we used the system to identify a number of bugs in Servo, including some resource leaks serious enough to lead to crashes.

3.7.1 Future Work

In the future we plan to use automatic weight adjusting techniques for performing Pairwise (or N-Way) feature testing. We also plan to explore other algorithms for weight selection, propagation, and reduction that may be even more effective and more efficient and to determine if there are more appropriate algorithms for certain types of grammars. Finally, the context-free nature of EBNF grammars limits the types of test cases that can be expressed so we plan to develop support for other types of formal grammars such as parsing expression grammars (PEGs) and to add a more granular annotation language for grammar weights such as the Geno language described by Lämmel [17].

Acknowledgments

Our thanks to Ingolf Krueger and Matt Oquist for their detailed feedback. Joel Martin thanks his supervisors at Viasat, Markus Kaltenbach and Ingolf Krueger, for their support and encouragement of his graduate studies and research. He also thanks his employer, Viasat Inc, for financially supporting his education. Any opinions expressed are those of the authors and not that of the institutions with which they are affiliated.

Chapter 4

Instacheck In Greater Depth

This chapter provides more detailed information about the Instacheck module (available at `https://github.com/kanaka/instacheck`) that is described in Chapter 3. In addition, more detail is provided about the Clojure PBT *test.check* library that Instacheck builds on.

4.1 Clojure test.check PBT Example

Section 3.3 provides an abstract example of defining PBT input and output properties and using PBT to check them. This section provides a more concrete example of testing the same SUT using Clojure's [11] *test.check* library [12]. Listing 4.1 shows the definition of a simple generator using both scalar and compound generators. The outer-most generator is `vector` which is a compound generator that returns zero or more values from the inner generator. The inner generator is `one-of` which is also a compound generator that selects

```
(def gen (vector (one-of [int double])))
```

Listing 4.1: Example of *test.check* generator (input property) composed of two compound generators and two scalar generators.

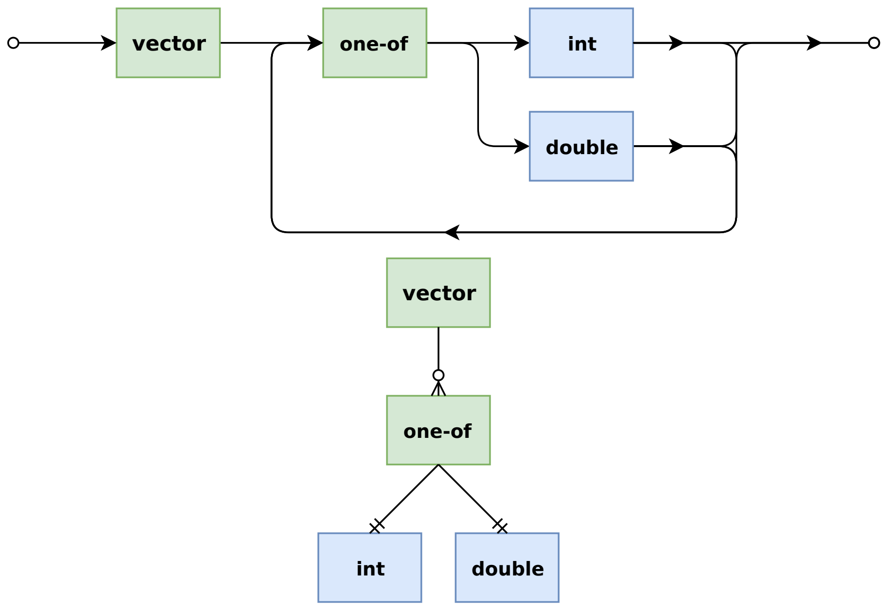


Figure 4.1: A railroad diagram and entity diagram representation of Listing 4.1

```
(take 10 (sample-seq gen))
;;=> ([] [0] [-1 -1.0] [1.5] [-1.5 1] [4] [2] []
;;     [3.1171875 -8 1.0 0.55078125 0.99609375 -2]
;;     [-1 3 0 0.0 -0.2880859375])
```

Listing 4.2: Example of values sampled from the generator defined in listing 4.1.

```
(defn oracle-fn [x]
  (not (some #(apply < %) (partition 3 1 x))))
(oracle-fn [7 2.1 -1 3 2])
;;=> true
(oracle-fn [7 -2.1 -1 3 2])
;;=> false
```

Listing 4.3: Example of *test.check* test Oracle (output property) function which returns false if 3 adjacent numbers are an increasing order

```
(quick-check 1000 (for-all* [gen] oracle-fn))
;;=> {:pass? false,
;;    :num-tests 7,
;;    :failing-size 6,
;;    :seed 1565579590402,
;;    :fail [[0.9140625 1.9375 -6 -6 -1 2.25]],
;;    :shrunk {:pass? false,
;;             :total-nodes-visited 46,
;;             :smallest [[-1 0 1]]}}
```

Listing 4.4: Example of results of using *test.check*'s *quick-check* function with the input property generator from listing 4.1 and the output property oracle function from listing 4.3

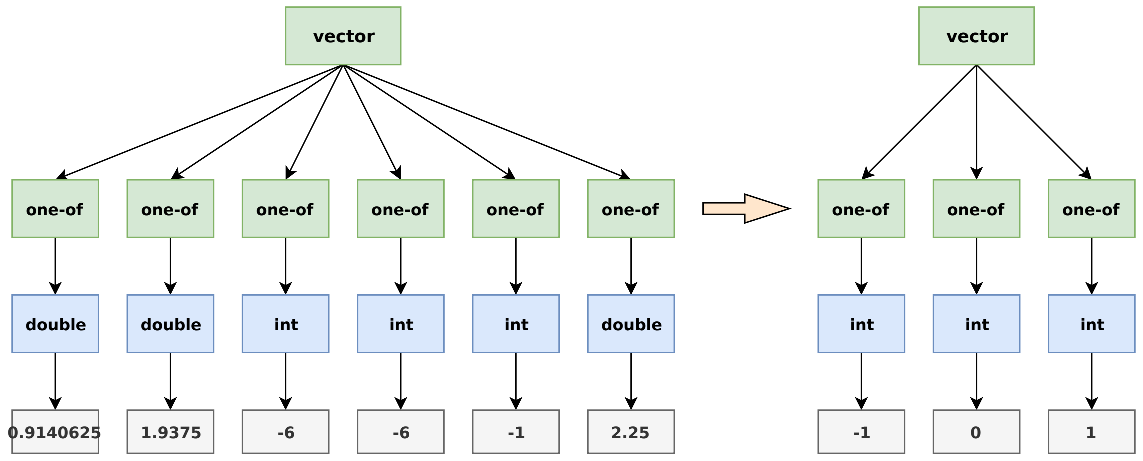


Figure 4.2: Test cases trees for the initial detected failure and failure after shrinking.

one of its inner generators to return a value. The inner generators are `int` which returns an integer value and `double` which returns a floating point value. The overall effect of the *gen* generator is to return a vector containing integers and floating point numbers. Figure 4.1 shows an entity relationship and railroad diagram representation for the *gen* generator.

We can now use the helper function `sample-seq` which returns an infinite lazy sequence of values from the given generator. Listing 4.2 shows the result of taking the first 10 values from an invocation of `sample-seq`. The result is 10 vectors each containing zero or more integers and floating point numbers. Each sample is generated using a larger size value than the previous sample. This means the maximum possible length of each vector increase, the maximum absolute magnitude of the integers increase, and the maximum absolute magnitude and precision of the floating point numbers increases with each subsequent sample.

We define an output property oracle function that returns false (counterexample) if the vector contains three adjacent numbers which are in increasing size order; otherwise, the function returns true (success). Listing 4.3 shows an example definition of a test oracle function and the result of using that function on a non-failing example test case and on a failing (counterexample) test case.

Listing 4.4 shows the result of running `quick-check` using the input property (`gen`) and output property (`oracle-fn`). The output shows that a failing test case was found on test iteration *7* with a generator size value of *6*. The sequence that caused the initial failure was `[0.9140625 1.9375 -6 -6 -1 2.25]`. The shrinking process then considered *46* nodes of the generator tree and returned `[-1 0 1]` as the smallest value that still fails the oracle check.

Figure 4.2 shows a tree representation of the initial failure test case and of the shrunk test case. The left side of Figure 4.2 gives a representation of the in-memory generator tree for the initial failing test case generated by *gen*. This test case violates the property defined in 4.3 that there should be no three adjacent numbers in ascending order due to the sequence [-6, -1, 2.25]. The right side of 4.2 shows the final shrunk test case case that continues to violate the property.

4.2 Clojure test.check Generators

The distribution of values returned by most *test.check* generators is not uniform. The frequency of smaller or simpler values is greater than the frequency of larger or more complex values. This distribution represents a tension between the goal of discovering simple test case counterexamples early in the testing process and the goal of still being able to discover more complex test counterexamples with increased iterations [33]. The rest of this section describes a few of the generators provided by the Clojure *test.check* library: *int*, *double*, *vector*, and *frequency*.

Figure 4.3 shows the distribution of values returned by the *test.check int* generator for different max size settings. For each maximum size setting, 1,000,000 samples were generated using the *int* generator. For a maximum size of 5, just over one 1/3rd values were equal to 0. A little less than 1/6th of the values were equal to -1 and 1/6th of the values were equal to 1. As the maximum size value is increased, the value generated become more evenly distributed but the numbers closer to 0 are always the most dominant.

Figure 4.4 show the distribution of values returned by the *test.check double* generator. The figure shows 100,000 values sampled from the generator with a maximum size of 80. Note that the X-axis in the figure is cropped to a minimum of -7 and maximum of 7 because the most relevant details related to distributed are in that range however there is a small

```
(frequency
  [[30  (return 1)]
   [10  (return 2)]
   [5   (return 3)]
   [30  (return 4)]
   [25  (return 5)]])
```

Listing 4.5: Example of a compound *test.check* frequency generator

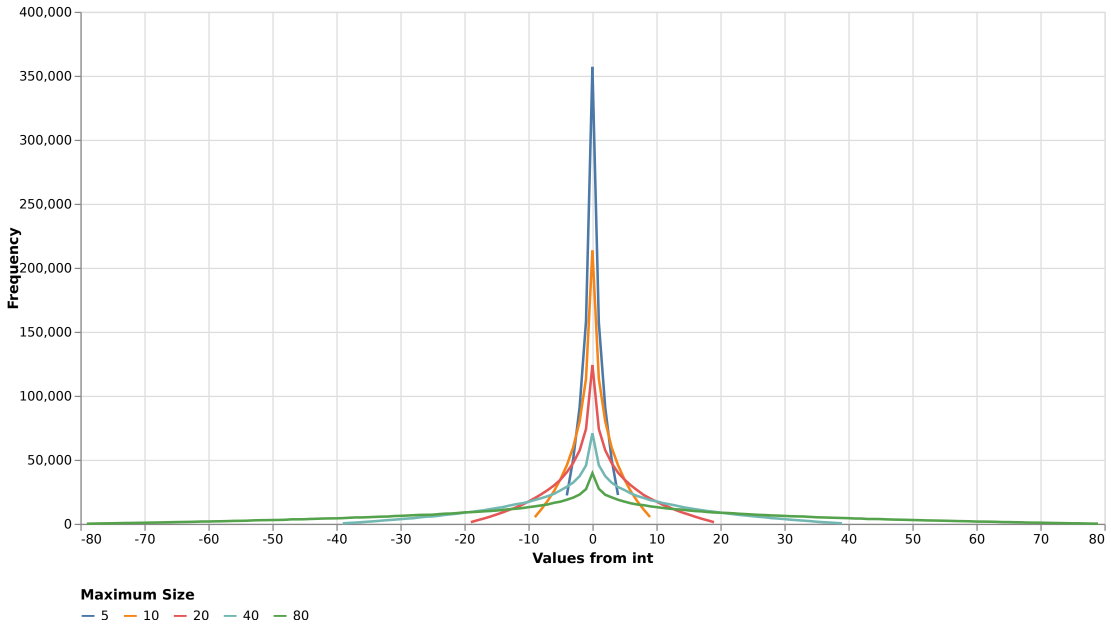


Figure 4.3: Values generated by the *int* generator sampled 1,000,000 times for each maximum size from 5 to 80

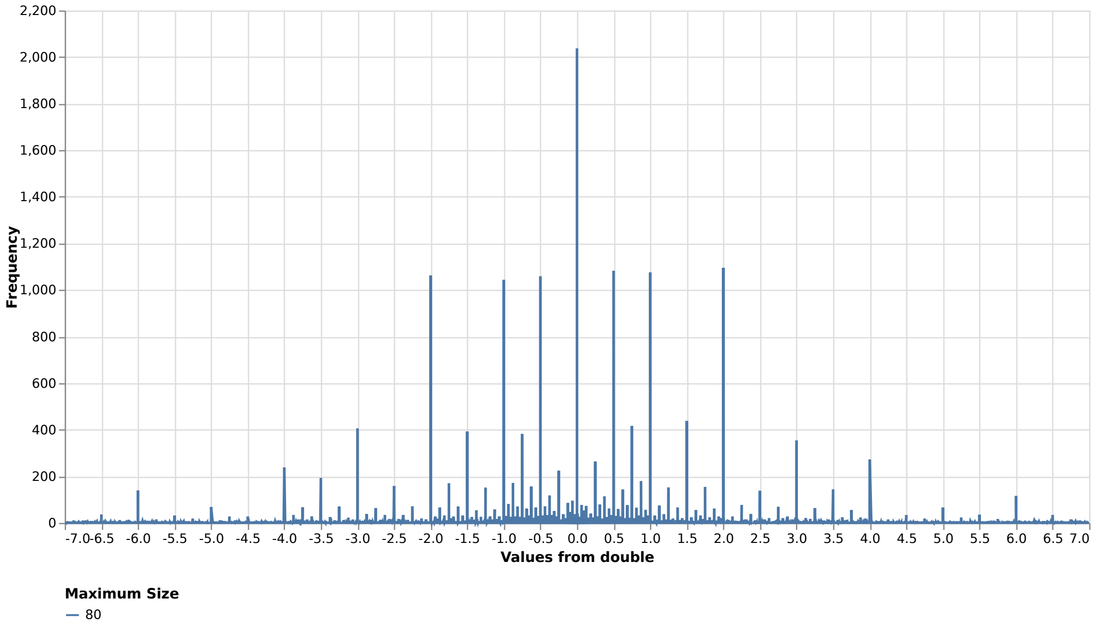


Figure 4.4: Values generated by *double* generator sampled 100,000 times with a maximum size of 80 and the X-axis cropped to between -7 to 7

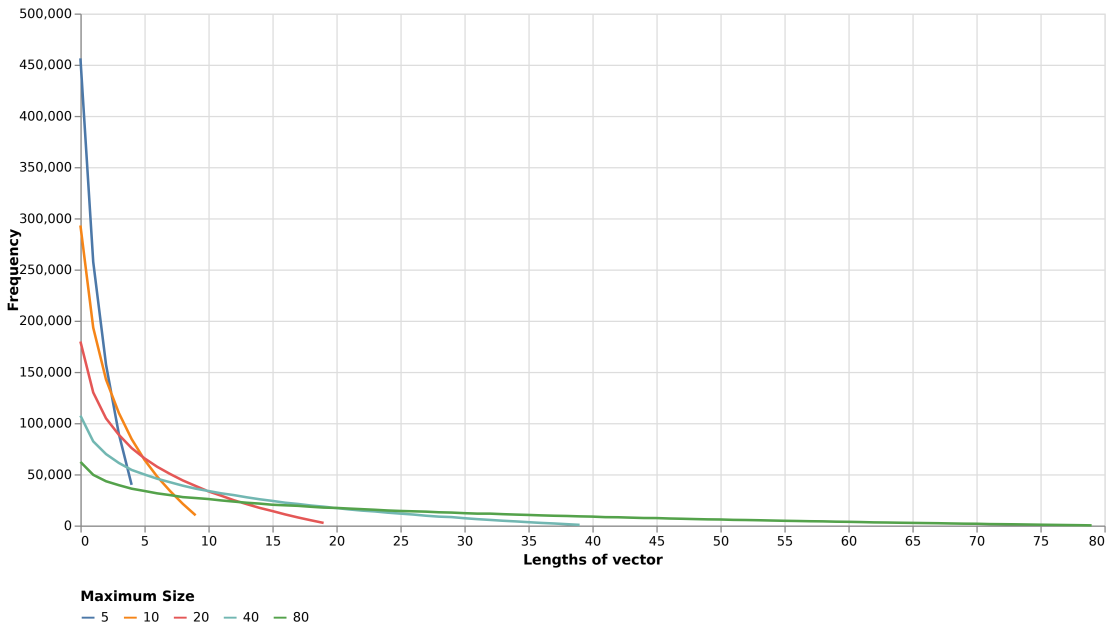


Figure 4.5: Length of vectors generated *vector* generator sampled 1,000,000 times for each maximum size from 5 to 80

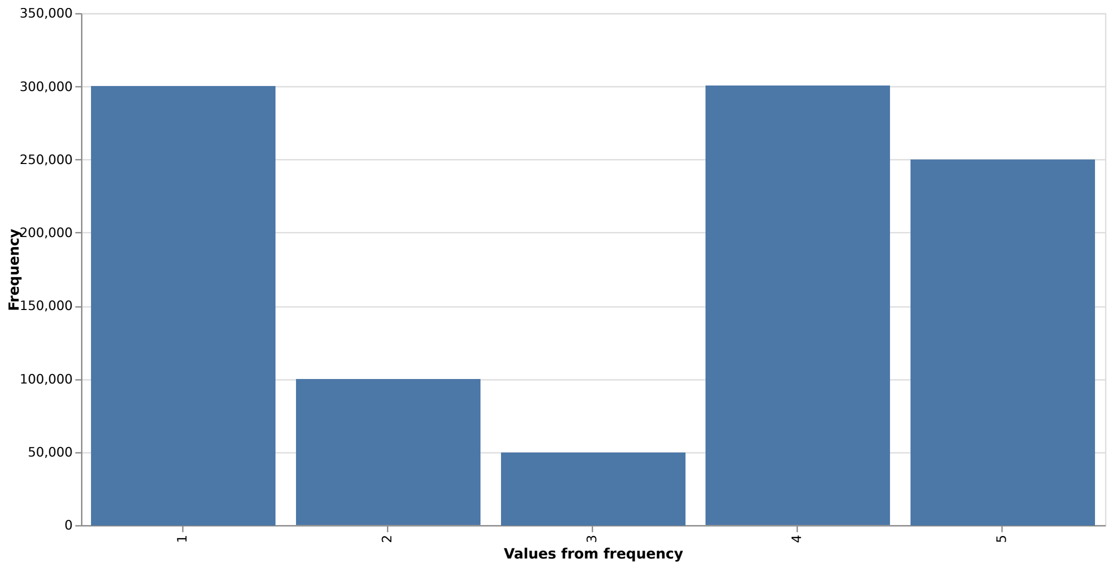


Figure 4.6: Values generated by *frequency* generator defined in Listing 4.5 sampled 1,000,000 times

distribution of values from -80 to 80 beyond the area shown in the figure. Similar to the *int* generator, 0.0 is the most frequently occurring value for the *double* generator and there is general drop in frequency of values as the magnitude increase. The degree of precision is an additional a measure of complexity/size for the *double* generator, . The next most common values after 0.0 are -2.0, -1.0, -0.5, 0.5, 1.0, and 2.0 are. In other words, the distribution of values from *double* is inversely correlated with both magnitude and level precision.

Figure 4.5 shows the distribution of vector lengths returned by the *test.check vector* generator (the contained generated returns a constant value). The most common vector length is 0 with larger sizes dropping off in frequency. Increasing the maximum size value flattens the distribution of lengths but the general distribution shape is maintained.

The *frequency* generator is a particularly important generator in the context of Instacheck because it is part of the mechanism used to implement the grammar weighting. Listing 4.5 shows the definition of a compound generator using *frequency*. The generator constructor takes a sequence of *[weight, generator]* tuples. The distribution of values from the *frequency* generator will match the weight defined for each value. In other words, the probability of a given value A being generated is:

$$P(A) = \frac{w(A)}{\sum_{a \in V} w(a)}$$

where w is a function returning the weight of a value and V is the set of all weights. Figure 4.6 shows the values generated from the generator defined in listing 4.5. Note that the weights in the generator sum to 100 to make it convenient to compare with the frequencies in figure. However, this is not necessary when defining a generator. For example, the value *1* occurs with a frequency of 30% (300,000 out of 1,000,000 samples).

```
(ebnf->gen
  "x = '1' | '2' | '3' | '4' | '5'")
```

Listing 4.6: Simple EBNF defined generator

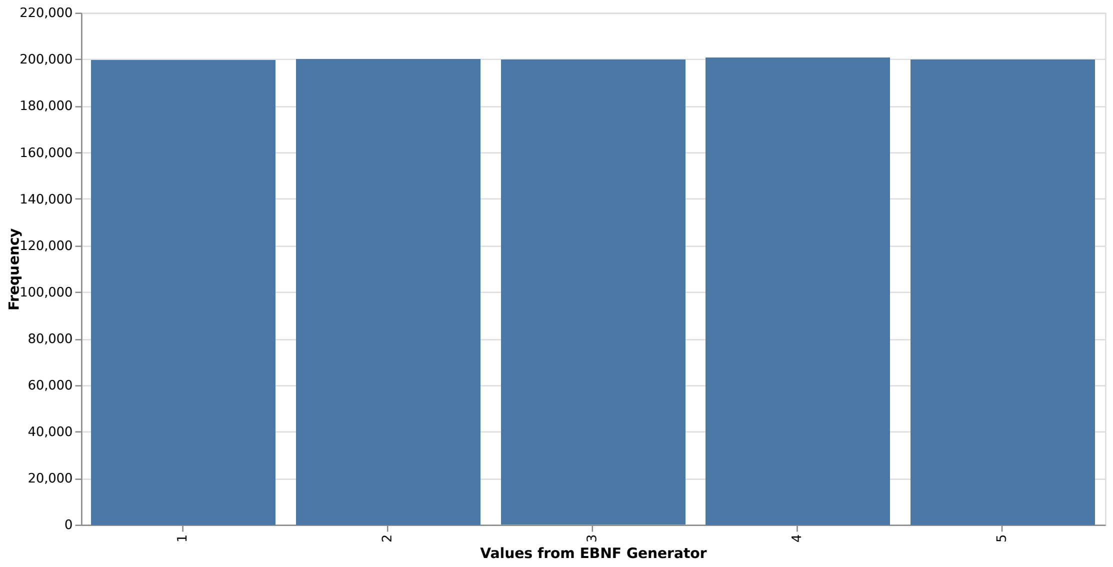


Figure 4.7: Values generated by the EBNF generator in Listing 4.6 sampled 1,000,000 times

4.3 EBNF Generators and Weight Tuning

The Instacheck library provides a convenience function `ebnf->gen` that takes an EBNF string and returns a generator based on that string. Listing 4.6 shows the definition of a simple generator using the EBNF definition `x = '1' | '2' | '3' | '4' | '5'`. The EBNF grammar has a single rule with an alternation that describes a string containing a single digit in the range 1 to 5 inclusive. Figure 4.7 shows frequency distribution of values from generator based on the EBNF grammar. The figure shows the result of sampling a value from the generator 1,000,000 times. Each value occurs approximately 20% of the time or about 200,000 occurrences.

The `ebnf->gen` function also has a two argument form that allows a configuration map to be provided that can adjust the behavior of the returned generator. Listing 4.7 shows the configuration map being used to specify user defined weights for each branch of the

```
(ebnf->gen
  {:weights {[:x :alt 0] 50
             [:x :alt 1] 25
             [:x :alt 2] 12
             [:x :alt 3] 8
             [:x :alt 4] 5}}
  "x = '1' | '2' | '3' | '4' | '5'")
```

Listing 4.7: Simple EBNF defined generator with user defined weights

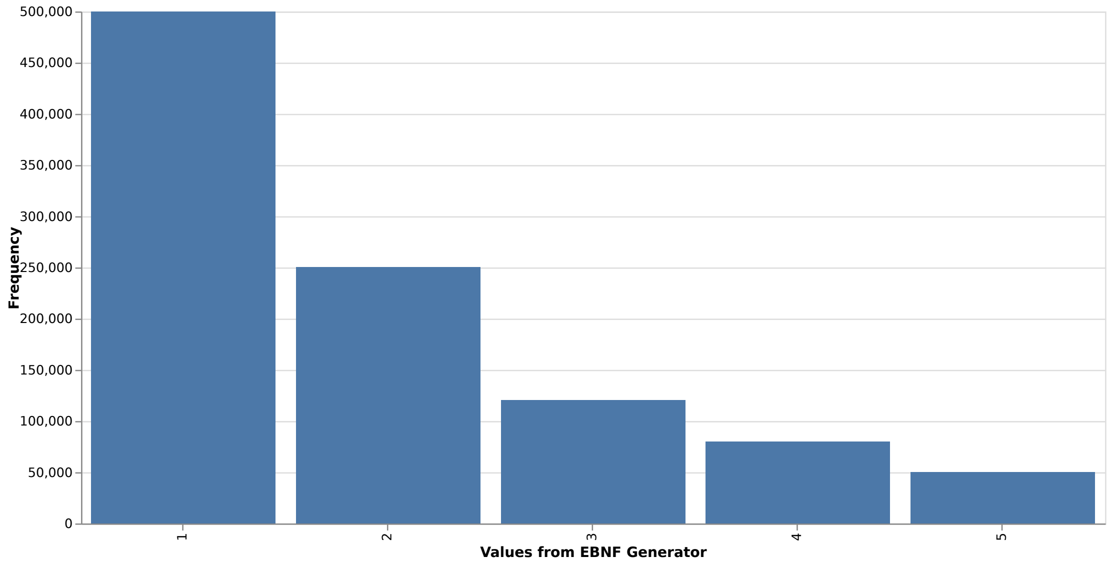


Figure 4.8: Values generated by the weighted EBNF generator in Listing 4.7 sampled 1,000,000 times

```
r1 = 'a' ( 'b' | #'[cd]+' | r2* )
r2 = 'e' r2 | 'f'
```

Figure 4.9: A grammar with two production rules.

```
("ad" "acc" "acc" "af" "ab" "ad" "accd" "adddcccdd"
 "ab" "aeeffffffeeff" "acdccdcdcddc" "adcdddd"
 "acdcccccd" "ab" "ab" "a" "addccdddd" "ab" "a"
 "aeefffefefefffffefeef" "acddcdccccc"
 "adccddddcdddcccdcdc" "a" "a"
 "adddcdccddcddcdcdcddcccdd" "ab" "a" "ab" "adc" "ab")
```

Listing 4.8: Example of 30 test cases sampled from a generator based on the grammar in Listing 4.9

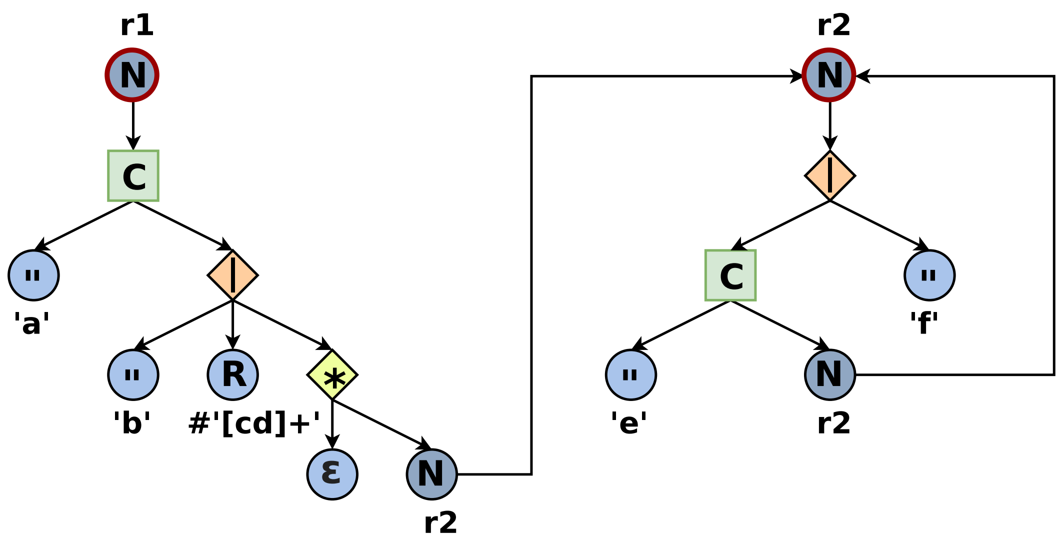


Figure 4.10: Graph representation of grammar in 4.9

```
{
 [:r1 :cat 1 :alt 0]             100,
 [:r1 :cat 1 :alt 1]             100,
 [:r1 :cat 1 :alt 2]             100,
 [:r1 :cat 1 :alt 2 :star nil]   100,
 [:r1 :cat 1 :alt 2 :star 0]     100,
 [:r2 :alt 0]                    100,
 [:r2 :alt 1]                    100
}
```

Listing 4.9: Default weights (wtrek) for the grammar in Listing 4.9

```
{
 [:r1 :cat 1 :alt 0]              10,
 [:r1 :cat 1 :alt 1]               0,
 [:r1 :cat 1 :alt 2]             100,
 [:r1 :cat 1 :alt 2 :star nil]     0,
 [:r1 :cat 1 :alt 2 :star 0]     100,
 [:r2 :alt 0]                    100,
 [:r2 :alt 1]                      0
}
```

Listing 4.10: Default weights (wtrek) for the grammar in Listing 4.9

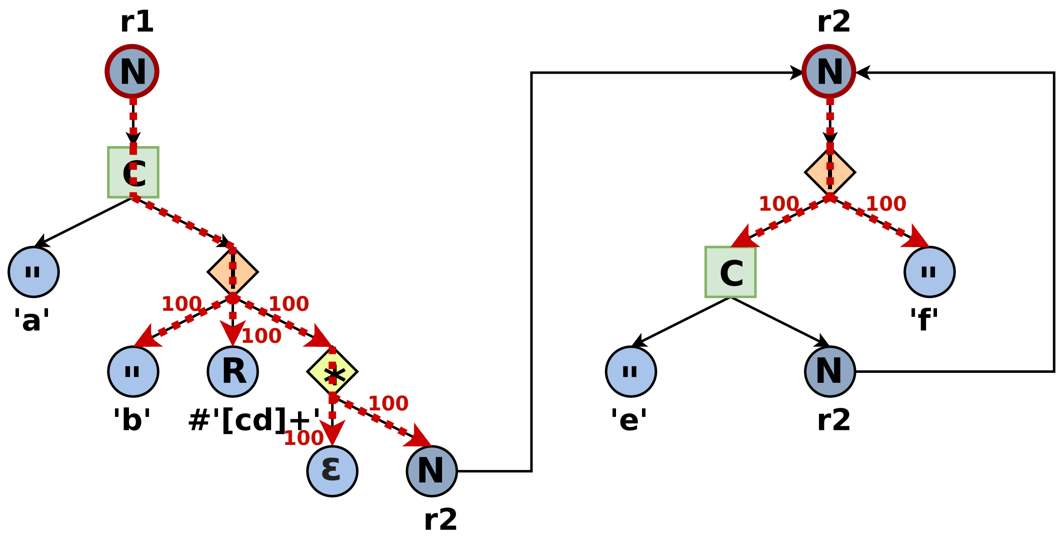


Figure 4.11: Graph representation of grammar in 4.9 annotated with weighted paths and default weights.

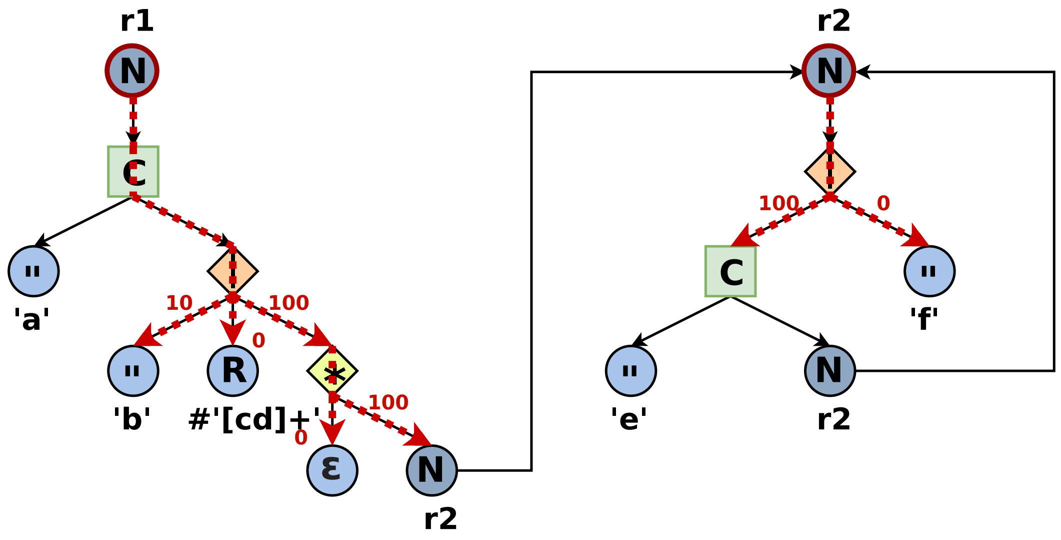


Figure 4.12: Graph representation of grammar in 4.9 annotated with weighted paths and adjusted weights.

```
("aef" "aefef" "aef" "aef" "af" "aefefef" "aefefeefeef"
 "aefefeffefefef" "aefeefffefef" "aef"
 "aeefeeffeefeefefefefefeefef" "aeefeeefeefeefeefeefeefef"
 "aeeffefeefeefeeeff" "aeefefefeefeeefeeffeefeefeef"
 "aeeefefeefeffef" "afefeefefefeefeefeeefefeefefeef"
 "aeeefeefeefefeeefef" "aefef" "afefeefeeef"
 "aeefeefefeefffeeefef" "ab" "afefeeefefefefeeefefefeeefeef"
 "aeefefef" "aeefeefef"
 "afefeefefeefeffefeefeefefefefefeefeefefeffeefeeefefef"
 "aeefeefeeefeeffeefeefeeefefeefeeefeefeefeefeef"
 "aefefeffefeeefeeefeeefeeefeefeeefefeefefeefeefeefeefeef"
 "ab" "ab" "ab")
```

Listing 4.11: Example of 30 test cases sampled from a generator based on the grammar in Listing 4.9 with weights adjusted with the wtrek in Listing 4.10

alternation in the grammar. The weights are used to adjust the frequency distribution for alternation rules in the grammar during generation of test cases. Figure 4.8 shows the effect that the weights have on the distribution of values returned from the generator.

Figure 4.9 shows a larger EBNF grammar that was originally introduced in Listing 3.3. In addition to containing two separate production rules, the grammar makes use of concatenation (C), alternation (◆), and zero-or-more (✱) combinators. It also contains three terminal types supported in Instacheck grammars: string literals ("), regular expression literals (R), and epsilon (ε). The epsilon terminal represents an empty or non-producing terminal type. Listing 4.8 shows an example of 30 test cases sampled from a generator based on the grammar.

The default weight that Instacheck assigns to each weighted path in a grammar tree is 100. For the grammar in Listing 4.9 this is equivalent to the weight trek (wtrek) shown in figure 4.9. All seven weighted paths in the grammar have a weight of 100. This is represented in visual form in Figure 4.11. The weighted paths in the grammar are depicted as red arrows from the start of the production to the weighted edge in the grammar. Each weighted edge show the edge weight in red. The default weight configuration is the one that was used to generate the samples shown in Listing 4.8.

Adjusting the wtrek configuration to the weights shown in Listing 4.10 will give the visual representation shown in Figure 4.12. Sampling the grammar generator using these adjusted weights will give a test result that resembles Listing 4.11. The samples are larger than with the default wtrek weight configuration because the overall probabilities of reaching and recursing back into the `:r2` rule are much greater in this configuration.

4.3.1 MG4 and MG8 Grammars

Section 3.5 introduced the MG4 and MG8 grammars that are used to validate the Instacheck approach. Listing 3.8 in that section shows the EBNF definition of the MG4 grammar. Figure 4.13 shows the visual representation of the MG4 grammar. Note that the representation only shows the grammar graph below the "tag1" alteration. The other tag alternations are elided for space but their portion of the graph is identical to the "tag1" portion of the graph. Listing 4.12 shows the MG8 variant of the grammar that has eight possible choices at each tag, attribute, property name, and property value alteration point in the grammar. Listing 4.13 shows the three results from an invocation of `sample-seq` using the MG4. These results have had indenting added to clarify the structure of the results.

The heat maps shown in Figure 4.16 depict the results of taking 100 random samples from the MG4 and MG8 grammar generators respectively and then mapping the TAPV features within those samples into boxes arranged by tag (large boxes on X-axis), attribute (small boxes on X-axis), property name (large boxes on Y-axis), and property values (small boxes on Y-axis). The yellow boxes indicate light coverage of those TAPVs and the darker blue indicates high coverage of those TAPVs. If the weight configuration is adjusted using the partial wtrek shown in figure 4.15 and then 100 new samples are taken from the grammar generator, the coverage will look similar to the heat map in 4.16.

```
test = elem+
elem = '<tag0' ( ' ' attr )+ '>' elem* '</tag0>'
     | '<tag1' ( ' ' attr )+ '>' elem* '</tag1>'
     | '<tag2' ( ' ' attr )+ '>' elem* '</tag2>'
     | '<tag3' ( ' ' attr )+ '>' elem* '</tag3>'
     | '<tag4' ( ' ' attr )+ '>' elem* '</tag4>'
     | '<tag5' ( ' ' attr )+ '>' elem* '</tag5>'
     | '<tag6' ( ' ' attr )+ '>' elem* '</tag6>'
     | '<tag7' ( ' ' attr )+ '>' elem* '</tag7>'
attr = aname '="' prop ( ' ' prop )* '"'
prop = pname ':' ( ' ' pval )+ ';'

aname = 'attr0'
      | 'attr1'
      | 'attr2'
      | 'attr3'
      | 'attr4'
      | 'attr5'
      | 'attr6'
      | 'attr7'
pname = 'pname0'
      | 'pname1'
      | 'pname2'
      | 'pname3'
      | 'pname4'
      | 'pname5'
      | 'pname6'
      | 'pname7'
pval  = 'pval0'
      | 'pval1'
      | 'pval2'
      | 'pval3'
      | 'pval4'
      | 'pval5'
      | 'pval6'
      | 'pval7'
```

Listing 4.12: Markup grammar degree 8 (MG8)

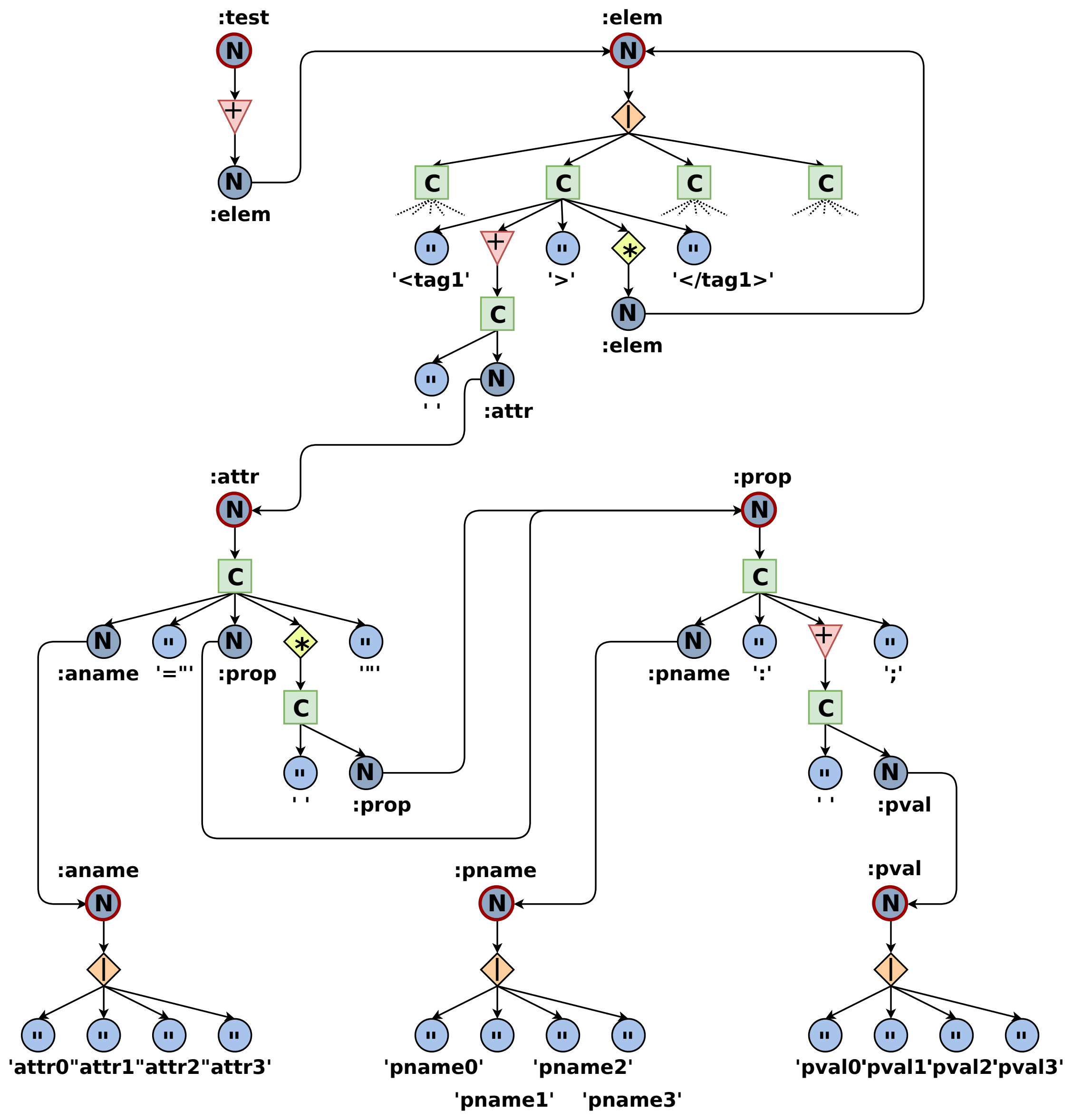


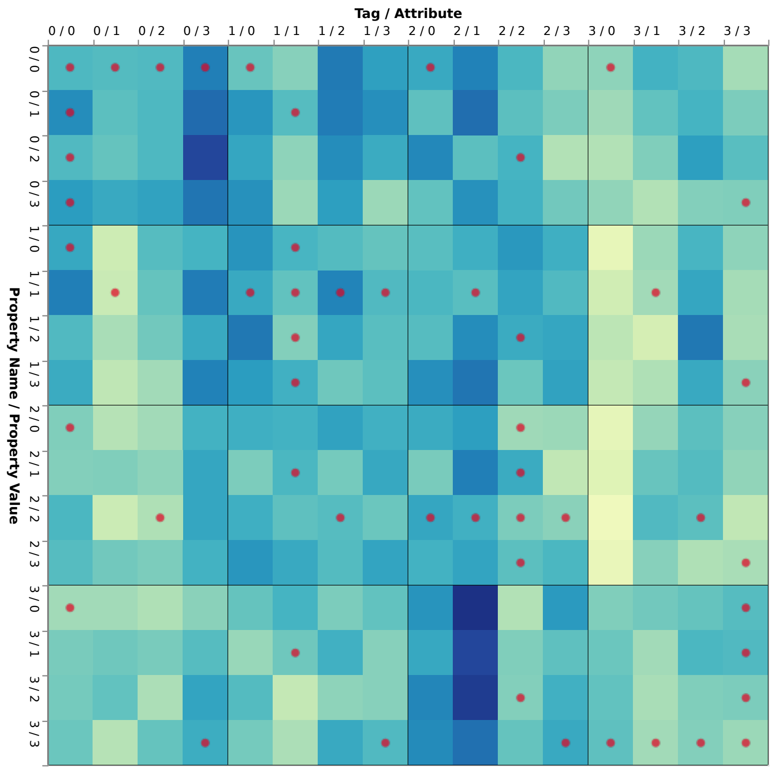
Figure 4.13: Visual representation of the MG4 grammar

```
<tag1 attr1="pname3: pval3;">
  <tag1 attr1="pname3: pval3;">
  </tag1>
</tag1>
```

```
<tag3 attr2="pname2: pval1;"
      attr1="pname0: pval1 pval3;">
</tag3>
```

```
<tag3 attr2="pname2: pval2;
             pname1: pval3;">
  <tag1 attr2="pname0: pval0;
               pname1: pval1 pval0 pval0;
               pname0: pval0 pval0 pval3;
               pname1: pval3;
               pname2: pval3 pval0;"
        attr3="pname0: pval1 pval3;
               pname0: pval3 pval0 pval1;"
        attr1="pname3: pval3 pval1 pval3;
               pname3: pval1 pval2 pval0 pval1;
               pname2: pval2 pval3 pval2 pval2;
               pname3: pval0 pval2;">
    <tag2 attr3="pname2: pval3;"
          attr0="pname0: pval3 pval1 pval0;
                 pname3: pval2 pval3;
                 pname0: pval2 pval3;
                 pname1: pval1;">
    </tag2>
  </tag1>
</tag3>
```

Listing 4.13: MG4 sample output with default weights



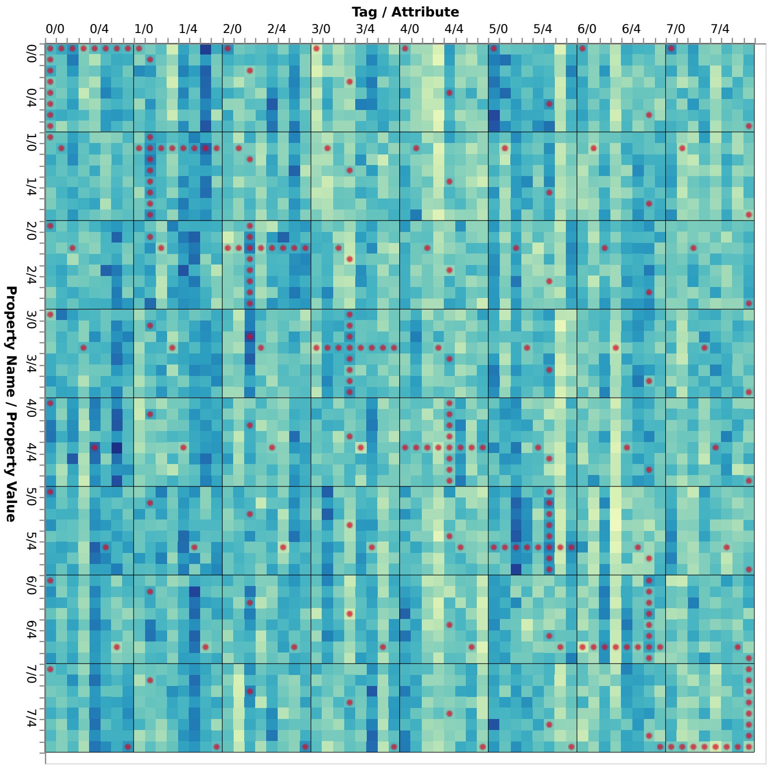


Figure 4.14: Coverage of the MG4 and MG8 grammars for 100 runs with default grammar weights. The X axis delimits tags (large boxes) and attributes (small boxes) and the Y axis delimits properties (large boxes) and property values (small boxes).

```
{
  [:elem :alt 2]  1000,
  [:elem :alt 3]  0,
  [:pname :alt 4] 2,
  [:aname :alt 5] 200
}
```

Figure 4.15: Partial wtrek of weight adjustments for the MG8 grammar

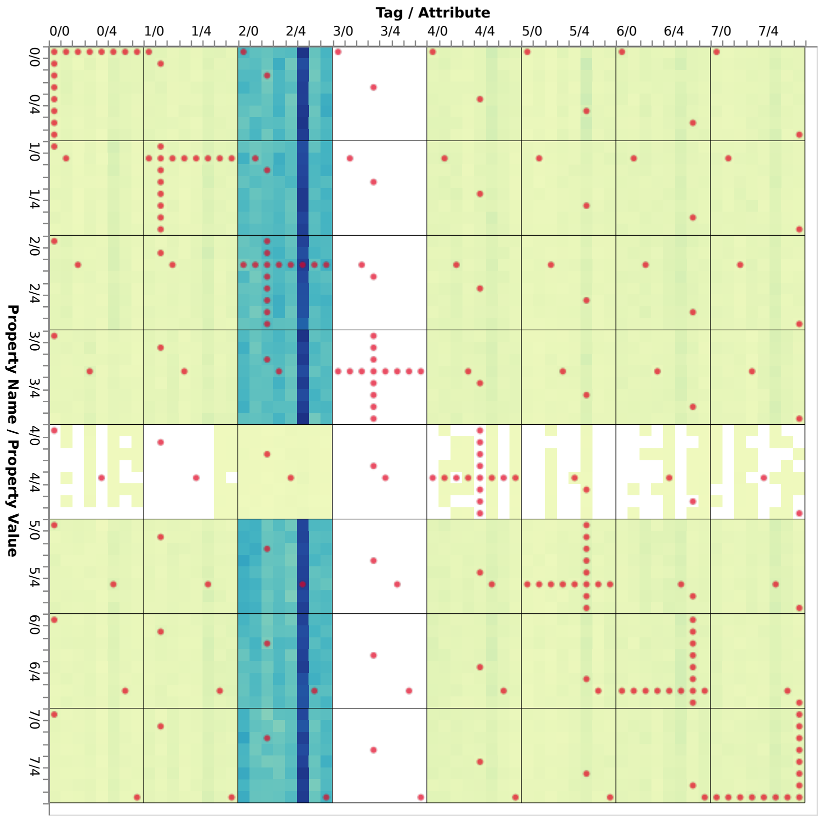


Figure 4.16: Coverage of the MG8 grammar for 100 runs with weights set to the partial wtrek shown in Listing 4.15. The X axis delimits tags (large boxes) and attributes (small boxes) and the Y axis delimits properties (large boxes) and property values (small boxes).

4.4 Translating EBNF Grammars to Clojure Generators

An EBNF grammar has a general graph structure because production rules can refer to other arbitrary production rules. However, each individual production rule of EBNF is a tree structure. Instacheck is able to take advantage of this structure by iterating through each production rule independently and using a recursive descent process to translate the content of a production rule. At each node of the rule Instacheck has a dispatch routine that calls the translation function for the given node type. The translation functions then call back into the dispatch function for each component of the combinator. Instacheck also passes an indent parameter as part of the recursive descent that is used to output code that is formatted for human readability.

Clojure's *test.check* library provides the *recursive-gen* function for defining recursive generators that return tree shaped data. Recursive generators must be defined using *recursive-gen* in order to guarantee that the generator eventually terminates. Generators that are defined in this way will have the a maximum number of leaf nodes that is proportional to the current generation size parameter. The *recursive-gen* generator takes two other generators as arguments. The first generator argument is the recursive generator that represent the inner structure of the tree structure that will be generated. The second generator argument will be used for the leaf nodes of the generated tree structure and so it must not be recursive. We will refer to these two arguments as the tree generator and leaf generator respectively. While the *recursive-gen* method of defining recursive generators allows recursive

```
x = 'a' 'x' | 'b' | 'c' ;
```

```
x = 'a' x | 'b' | 'c' ;
```

Listing 4.14: Simple non-recursive and recursive EBNF grammars

```
(igen/freq :x [
  [100
    (gen/tuple
      (gen/return "a")
      (gen/return "x"))]
  [100
    (gen/return "b")]
  [100
    (gen/return "c")]]))
```

Listing 4.15: Clojure translation of the simple non-recursive EBNF grammar in Listing 4.14.

```
(gen/recursive-gen
  (fn [inner]
    (igen/freq :x [
      [100
        (gen/tuple
          (gen/return "a")
          inner)]
      [100
        (gen/return "b")]
      [100
        (gen/return "c")]]))
  (igen/freq :x [
    [100
      (gen/return "b")]
    [100
      (gen/return "c")]]))
```

Listing 4.16: Clojure translation of the simple recursive EBNF grammar in Listing 4.14.

data to be generated without the problem of runaway production of data, it is inconvenient for automatic translation EBNF grammars. In addition, it does not provide a direct means of defining mutually recursive generators.

There are four challenges that must be addressed by Instacheck during translation: mutual recursion must be prevented, the tree generator must be transformed to use the leaf generator internally, the leaf generator must not refer to itself, and generator code should be translated in dependency order to avoid the necessity of forward references in the code.

Instacheck prevents mutual recursion by performing a topological sort of the grammar using Kahn's algorithm [34]. The presence of a non-terminal within a grammar rule is treated as a dependency edge between the contained non-terminal and the rule that it references. We modified a Clojure implementation [35] of Kahn's algorithm to add support for enumerating mutually recursive cycles in the dependency graph. Any mutual recursion detected is reported as an error during the EBNF to Clojure code translation process.

Instacheck transforms the tree generator to use the leaf generator parameter by tracking the current grammar rule name in a context parameter that is passed through to the recursive descent translation functions. When a non-terminal is encountered it is compared to the current rule name and if it matches then it is replaced with the leaf generator parameter.

Instacheck removes self-references within the leaf generator by identifying the smallest optional branches of the generator which are recursive and removing those branches. In this case, "optional" means that the branch is a weighted child edge of a parent node. Instacheck does this pruning by performing a postwalk on the grammar tree and progressively rewriting sub-trees of the grammar until a optional edge of the grammar is reached.

In addition to mutual recursion detection, we also use Kahn's algorithm to sort the grammar rules during translation so that they are translated in a dependency order. Before sorting, directly recursive references are pruned so that the topology sort can complete without error. This is acceptable for purposes of code order because recursive generators do not need forward references due to the Clojure property that functions can refer to themselves (as is generally true in any language that supports recursion).

Listing 4.14 shows two simple EBNF grammars. The first contains no recursion while the second replaces the terminal string "x" with a recursive reference to the rule. Listings 4.16 and 4.16 demonstrate the translated generator code for first and second grammars respectively. In the second translation the entire generator has been wrapped with a `gen/recursive-gen` generator and the `(gen/return "x")` generator has been replaced by a reference to the `inner` parameter of the recursive generator. In addition, the second argument to `gen/recursive-gen` is a version of the same translated code but with the recursive parts of the tree removed.

4.5 Instacheck Command Line and Library Usage

```
line = expression '\n'?
expression = "(" expression ")"
           | expression "+" expression
           | expression "-" expression
           | expression "*" expression
           | expression "/" expression
           | any-number
any-number = any-digit
           | nz-digit any-digit+
any-digit  = "0"
           | nz-digit
nz-digit   = "1"
           | "2"
           | "3"
           | "4"
           | "5"
           | "6"
           | "7"
           | "8"
           | "9"
```

Listing 4.17: EBNF representing a simple mathematical expression

One of the advantages of using a generic test specification language like EBNF is that tests can be written and executed without a need for tests to be written in a domain-specific test

```
(require '[instacheck.core :as instacheck])
```

Listing 4.18: Clojure namespace preamble for requiring Instacheck modules

```
lein run samples --samples 100 --weights-output tmp/weights.edn \
    math.ebnf tmp/
```

Listing 4.19: Generate 100 random test cases based on the EBNF grammar in Listing 4.17

specification language. Our implementation of the Instacheck system provides a command line interface for testing a command line program by using input tests specified via EBNF. The EBNF grammar depicted in Listing 4.17 is used for the command line examples described in this section. In addition, the command line examples use the Leiningen project [36] for Clojure project management and execution. Leiningen is invoked via the `lein` command.

In addition to command line usage, Instacheck also provides a library of functions for that loading and manipulating grammars, running PBT tests using those grammars, running the reduction algorithms directly, etc. The examples below demonstrate a small subset of the functions that are available. See Appendix B for a more complete list and description of available Instacheck library functions. Listing 4.18 shows the Clojure command to import (*require*) the Instacheck library for use in the examples that follow.

4.5.1 Generate Random Test Cases

Command Line

Instacheck can be invoked from the command line to generate random test cases based on an EBNF specification. Consider the EBNF definition in Listing 4.17 for a document containing a single simple mathematical expression.

Given a file `math.ebnf` containing the EBNF grammar in Listing 4.17, then random test samples can be generated that satisfy the grammar as shown in listing 4.19. Examples of

```
0*60101403+8
```

```
40600080410*3905/(907590727-7)/(134940050403203641/4008350)
```

```
((0))-(3948502540020)+0+0
```

Listing 4.20: Test cases that satisfy the EBNF grammar in Listing 4.17

```
{
 [:any-digit :alt 0]       100,
 [:any-digit :alt 1]       100,
 [:any-number :alt 0]      100,
 [:any-number :alt 1]      100,
 [:expression :alt 0]      100,
 [:expression :alt 1]      100,
 [:expression :alt 2]      100,
 [:expression :alt 3]      100,
 [:expression :alt 4]      100,
 [:expression :alt 5]      100,
 [:line :cat 1 :opt 0]     100,
 [:line :cat 1 :opt nil]   100,
 [:nz-digit :alt 0]        100,
 [:nz-digit :alt 1]        100,
 [:nz-digit :alt 2]        100,
 [:nz-digit :alt 3]        100,
 [:nz-digit :alt 4]        100,
 [:nz-digit :alt 5]        100,
 [:nz-digit :alt 6]        100,
 [:nz-digit :alt 7]        100,
 [:nz-digit :alt 8]        100
}
```

Listing 4.21: Default active wtrek for the EBNF grammar in Listing 4.17

```
400717-5770
```

```
7670000707070770000+1/(760873724720779407070)
```

```
7/7+9/0/97-77779077131870870137200000090707077074700077+7075907070
```

Listing 4.22: Test cases that satisfy the EBNF grammar in Listing 4.17 and that have the weight of digit “7” increased to 1000.

sample files that are generated in the `tmp/` directory are shown in Listing 4.20.

The command line option `--weights-output` is used to create the file `tmp/weights.edn` containing the active wtrek that was used during the sample generation (shown in Listing 4.20). This file can be manually modified to adjust the active wtrek configuration for subsequent runs and loaded with the `--weights` option. For example, if the weight for the grammar path `[:nz-digit :alt 6]` is increased to 1000 then the generated test cases will show an increase in the frequency of the digit “7” as shown in listing 4.22

Library

Instacheck provides a `ebnf-sample-seq` function that can be used to generate test cases from an EBNF specification. An example of this is depicted in Listing 4.17. The function takes an EBNF string and an optional configuration. The configuration in the example is used to decrease the probability of the `[:x :start nil]` path through the grammar. This path represents the empty or epsilon case of the outer zero-or-more (“*”) grammar element. The effect is to increase the length of generated strings. The `ebnf-sample-seq` function returns an infinite lazy sequence of samples of increasing size. The `take` function is used to realize and return the first 20 elements of the sequence. Listing 4.17 shows the resulting samples that are returned by the command.

```
(take 20
  (instacheck/ebnf-sample-seq
    "x = ( '1' | '2' | '3' | '4' | '5' )* ;"
    {:weights {[:x :star nil] 10}}))
```

Figure 4.17: Generate 20 random test cases based on a simple EBNF grammar using the *ebnf-sample-seq* library function

```
("4" "52" "" "54" "511" "115533" "221341" "5113444" "341111235"
 "" "123121" "" "22" "2" "51245255314452" "3422332422313424"
 "2253212432" "44443" "4151332153351" "45123533141145")
```

Figure 4.18: Result of generating 20 random test case samples using the *ebnf-sample-seq* function as shown in Listing 4.17

4.5.2 Test and Shrink

Command Line

Instacheck can be invoked from the command line to run the full PBT testing process against another command line program that serves as a SUT test executor and test oracle. The simple shell script shown in Listing 4.23 evaluates the mathematical input expression using python and ruby as SUT and test oracle respectively. Listing 4.24 shows the command used to invoke Instacheck's PBT check process. The % symbol in the command line is a special placeholder that is replaced with the path to the test case for the current test iteration. Listing 4.25 shows the result of the check process. The check process ran 10 check iterations and did not find any failures where the python and ruby expression evaluation differed.

When the number of iterations is increased to 50 with the `--iterations` option (listing 4.26) then a failing test case is found and shrunk as shown in Listing 4.27 ("..." represents elided lines from the output). The initial failure that was found on the 36th iteration was `80208600000000402010-0/90400205/601700706199307000000 6817`. The shrink process then activated and after 221 further iterations determined that `100020063008500036100/6` was a

```
#!/bin/bash

set -e

SUT() {
  python -c "from __future__ import division; print '%d' % (${1})"
}
ORACLE() {
  ruby -e "require 'mathn'; printf '%d', (${1})"
}

res=$(SUT $(cat "${1}"))
check=$(ORACLE $(cat "${1}"))

echo "SUT: ${res}, Oracle: ${check}"
[[ "${res}" == "${check}" ]]
```

Listing 4.23: A simple SUT and test oracle script

```
lein run check-once math.ebnf tmp/ -- mathtest.sh %
```

Listing 4.24: Check EBNF grammar in Listing 4.17 using mathtest.sh SUT/Oracle harness in Listing {reflst:mathtest.sh

```
Running: ./mathtest.sh tmp/sample-0001
Result: Pass
NEW STATE: trial
Running: ./mathtest.sh tmp/sample-0002
Result: Pass
Running: ./mathtest.sh tmp/sample-0003
Result: Pass
...
Running: ./mathtest.sh tmp/sample-0009
Result: Pass
Running: ./mathtest.sh tmp/sample-0010
Result: Pass
NEW STATE: complete
Saving weights to tmp/weights.edn
Saving result map to tmp/result.edn
Result:
{:result true,
 :pass? true,
 :num-tests 10,
 :time-elapsed-ms 892,
 :seed 1570598045791}
```

Listing 4.25: Result of 10 check iterations of the EBNF grammar in Listing 4.17 with elided lines

```
lein run check-once math.ebnf tmp/ --iterations 50 -- ./mathtest.sh %
```

Listing 4.26: Run 50 check iterations of the EBNF grammar in Listing 4.17 using mathtest.sh SUT/Oracle harness in Listing 4.23

```
Running: ./mathtest.sh tmp/sample-0001
Result: Pass
NEW STATE: trial
Running: ./mathtest.sh tmp/sample-0002
Result: Pass
...
Running: ./mathtest.sh tmp/sample-0035
Result: Pass
Running: ./mathtest.sh tmp/sample-0036
Result: Fail (exit code 1)
NEW STATE: failure
Running: ./mathtest.sh tmp/sample-0037
Result: Fail (exit code 1)
NEW STATE: shrink-step
Running: ./mathtest.sh tmp/sample-0038
Result: Pass
...
Running: ./mathtest.sh tmp/sample-0257
Result: Fail (exit code 1)
NEW STATE: shrunk
Saving weights to tmp/weights.edn
Saving result map to tmp/result.edn
Result:
{:shrunk
 {:total-nodes-visited 213,
  :depth 17,
  :pass? false,
  :result false,
  :result-data nil,
  :time-shrinking-ms 18804,
  :smallest ["10002006300850036100/6"]},
 :failed-after-ms 3096,
 :num-tests 36,
 :seed 3,
 :fail ["8020860000000402010-0/90400205/60170070619930700000006817"],
 :result false,
 :result-data nil,
 :failing-size 35,
 :pass? false}
Smallest Failure: tmp/sample-final
```

Listing 4.27: Result of 50 check iterations of the EBNF grammar in Listing 4.17 with elided lines

simpler test case that still reproduced the problem. Note: the reason for the discrepancy between Python and Ruby is because the `from __future__ import division` changes the default behavior of integer division to always return a real number rather than an integer. This results in different precision from ruby during the conversion back to an integer by the print statement.

Library

The Instacheck function `instacheck` can be used to perform QuickCheck PBT testing using EBNF grammars. The second argument to `instacheck` is an EBNF test case specification. The first argument is the function that the user wishes to check. This check function takes a single argument and is called repeatedly by the test system with input strings generated from the EBNF grammar. When the function returns a *false* or *nil* value, then the test system treats the input as a failing test case (counterexample). Otherwise, the input is considered passing.

Listing 4.19 shows a simple example of how the command can be used. The check function uses a simple regular expression match and will return nil if the test case contains two adjacent "3" digits. The EBNF grammar is a simple grammar that defines test cases contain zero or more of the digits "1" through "5". Listing 4.20 shows two example results of that simple test invocation. The first result is a test run where 10 iterations were performed (10 is the default) and no failures were identified. The second result is a test run where a failing test case, "54133" was identified on iteration 6. The shrinking process activated and executed more iterations until it determined that "33" was the smallest test case that still

```
(instacheck/instacheck
  (fn [t] (not (re-seq #"33" t)))
  "x = ( '1' | '2' | '3' | '4' | '5' )* ;")
```

Figure 4.19: Check (test) a function with a simple EBNF grammar using the *instackeck* library function

```
{:result true,
 :result-data nil,
 :pass? true,
 :num-tests 10,
 :time-elapsed-ms 2,
 :seed 1573679260619}
```

```
{:result false,
 :result-data nil,
 :pass? false,
 :num-tests 6,
 :failing-size 5,
 :failed-after-ms 1,
 :seed 1573679262126,
 :fail ["54133"],
 :shrunk {:total-nodes-visited 10,
          :depth 2,
          :pass? false,
          :result false,
          :result-data nil,
          :time-shrinking-ms 3,
          :smallest ["33"]}}
```

Figure 4.20: Result of checking a function with an EBNF grammar as shown in Listing 4.19

had a failing result.

The two arguments to the `instacheck` function are Instacheck analogs for the input and output properties of standard PBT; the EBNF grammar string defines the input property and the check function is the output property. In actual practice, the check function will often be more sophisticated than the simple example shown. For example, a typical check function will do the following: deserialize the test case string into a form that can be used to invoke the SUT; invoke the SUT using the test case; and then invoke a test oracle to determine if the SUT behavior was correct for the given test case.

4.5.3 Reproduce and Shrink an existing Test Case

Command Line

Listing 4.28 shows an existing test case that causes the mathtest.sh harness in listing 4.23 to return a failure due a difference in result between the Python SUT and the Ruby test oracle. Instacheck can be invoked from the command line to parse an existing test case to extract a path log frequency trek (grammar path frequency log). The command to do this is shown in Listing 4.29 and the resulting path log frequency trek file `test1.edn` is shown in Listing 4.30.

The path log frequency trek can then be used directly as the active trek configuration for Instacheck to focus test coverage on the elements contained in the test case in order to reproduce similar failing test cases. Once a failing case is found, Instacheck applies the shrinking process. The command to run Instacheck with the parsed path log frequency trek is shown in Listing 4.31.

```
1000000000000000002/2
```

Listing 4.28: An existing test case that causes the mathtest.sh harness in Listing 4.23 to return a failure

```
lein run parse math.ebnf test1.expr > test1.edn
```

Listing 4.29: Parse a path log frequency trek from existing test case

```
{
 [:any-digit :alt 0]       16,
 [:any-digit :alt 1]       2,
 [:any-number :alt 0]      1,
 [:any-number :alt 1]      1,
 [:expression :alt 0]      0,
 [:expression :alt 1]      0,
 [:expression :alt 2]      0,
 [:expression :alt 3]      0,
 [:expression :alt 4]      1,
 [:expression :alt 5]      2,
 [:line :cat 1 :opt 0]     1,
 [:line :cat 1 :opt nil]   0,
 [:nz-digit :alt 0]        1,
 [:nz-digit :alt 1]        2,
 [:nz-digit :alt 2]        0,
 [:nz-digit :alt 3]        0,
 [:nz-digit :alt 4]        0,
 [:nz-digit :alt 5]        0,
 [:nz-digit :alt 6]        0,
 [:nz-digit :alt 7]        0,
 [:nz-digit :alt 8]        0
}
```

Listing 4.30: Parsed path log frequency trek from existing test case

```
lein run check-once math.ebnf tmp/ --weights test1.edn \
    --iterations 50 -- ./mathtest.sh %
```

Listing 4.31: Run 50 check iterations of the EBNF grammar in Listing 4.17 using mathtest.sh SUT/Oracle harness in Listing 4.23 and using parsed path log frequency trek in Listing 4.30

Library

The Instacheck `parse-wtrek` function can be used to parse a path frequency log trek from an existing test case. Listing 4.21 shows an example of invoking that function. The first argument to the function is a Instaparse parser which can be generated using the Instacheck `load-parser` convenience function. The second argument to the function is the test case string to parse, in this case "334". Listing 4.22 shows the result that is returned from parsing. The `:parsed` key contains the Instaparse parsed abstract syntax tree (AST). The `:wtrek` key contains a wtrek structure where the weights represent the frequency that the given path in the grammar was used to parse the test case.

The wtrek shown in Listing 4.22 that was returned by the `parse-wtrek` call in Listing 4.21 can be used to generate test cases that are similar to the test that was parsed. Generated test cases will be similar in the sense that the probability of elements in generated test cases will be similar to the frequency of those elements in the parsed test case. This can be used to reproduce an existing failing test case and then shrink it. The test case that was parsed, "31313143433411" is 14 character long and contains the failure condition (two adjacent "3" digits). The final shrunk case found by the `instacheck` function was "33".

```
(def parse-result
  (instacheck/parse-wtrek
    (instacheck/load-parser "x = ( '1' | '2' | '3' | '4' | '5' )* ;")
    "31313143433411"))
```

Figure 4.21: Use an EBNF defined parser to parse a path log frequency trek from an existing test case using the *parse-wtrek* library function

```
{:parsed [:x "3" "1" "3" "1" "3" "1" "4" "3" "4" "3" "3" "4" "1" "1"],
 :wtrek
 {[:x :star nil] 13,
  [:x :star 0] 14,
  [:x :star 0 :alt 0] 5,
  [:x :star 0 :alt 1] 0,
  [:x :star 0 :alt 2] 6,
  [:x :star 0 :alt 3] 3,
  [:x :star 0 :alt 4] 0}}
```

Figure 4.22: Result of parsing a path log frequency trek from an existing test case as shown in Listing 4.21

```
(instacheck/instacheck
  (fn [t] (not (re-seq #"33" t)))
  "x = ( '1' | '2' | '3' | '4' | '5' )* ;"
  {:weights (:wtrek parse-result)})
```

Figure 4.23: Check (test) a function with a simple EBNF grammar using the *instackeck* library function

```
{:result false,
 :result-data nil,
 :pass? false,
 :num-tests 3,
 :failing-size 2,
 :failed-after-ms 0,
 :seed 1573685725439,,
 :fail ["333"],
 :shrunk
 {:total-nodes-visited 3,
  :depth 1,
  :pass? false,
  :result false,
  :result-data nil,
  :time-shrinking-ms 1,
  :smallest ["33"]}}
```

Figure 4.24: Result of checking function with an EBNF grammar as shown in Listing 4.23

4.5.4 Translate EBNF Specification into Clojure Generators

Command Line

Instacheck can be invoked from the command line to translate an EBNF test specification directly to Clojure `test.check` generator code. This process is normally performed by Instacheck when it loads an EBNF test specification for testing. If Instacheck is used as a library within Clojure code, then it is much more efficient to translate the EBNF grammar once and load it directly as Clojure code. The command to load and translate an EBNF test specification is shown in listing 4.32. The second argument is the name of the Clojure namespace used when generating the code. Listing 4.33 shows a portion of the translated code for the `expression` generator. Listings A.1 and A.2 of Appendix A show the full translation of the grammar with each grammar rule defined as individual Clojure generators. The command also takes an optional `--function` argument that changes the mode so that a factory function, `gen-math` is generated rather than individual generator functions. The factory function is called to get the generator for a specific start rule. The factory function also takes an optional configuration parameter that allows the default weights to be adjusted. Listings A.1 and A.2 of Appendix A show the Clojure code output from the second mode.

```
lein run clj math.ebnf math.generators > generators.clj
```

```
lein run clj math.ebnf math.generators --function gen-math > generators.clj
```

Listing 4.32: translate the EBNF grammar in Listing 4.17 into a Clojure source file containing the equivalent `test.check` generator code.

```
...
(def gen-expression
  (gen/recursive-gen
    (fn [inner]
      (igen/freq :expression [
        [100 (gen/tuple (gen/return "(") inner (gen/return ")"))]
        [100 (gen/tuple inner (gen/return "+") inner)]
        [100 (gen/tuple inner (gen/return "-") inner)]
        [100 (gen/tuple inner (gen/return "*") inner)]
        [100 (gen/tuple inner (gen/return "/") inner)]
        [100 gen-any-number]]))
    gen-any-number))
...
```

Listing 4.33: Clojure generator code translated from the EBNF grammar in Listing 4.17 for the *expression* rule (reformatted)

Library

The `grammar->generator-defs-source` Instacheck library function translates an Instacheck grammar to Clojure generator code. Listing 4.25 shows an invocation of this function and Listing 4.26 shows the string that is returned with the equivalent translated Clojure `test.check` generator code. Note that the initial parameter to the call is a configuration option that contains default weights to use during the translation. In the translated code the modified weight is marked with the `;; ** adjusted by config ***` comment to make it clear that the default weight was explicitly adjusted by the caller.

```
(instacheck/grammar->generator-defs-source
  {:weights {[:x :star 0 :alt 0] 0}}
  (instacheck/load-grammar "x = ( '1' | '2' | '3' | '4' | '5' )* ;"))
```

Figure 4.25: Translate EBNF grammar to Clojure generator source using Instacheck

```clojure
(def gen-x
  (gen/fmap util/flatten-text
    (igen/freq :x [
      [100
        (gen/return "")]
      [100
        (igen/vector+
          (igen/freq :x [
            [0    ;; ** adjusted by config ***
              (gen/return "1")]
            [100
              (gen/return "2")]
            [100
              (gen/return "3")]
            [100
              (gen/return "4")]
            [100
              (gen/return "5")]]))]])))
```

Figure 4.26: Result of translating EBNF grammar in Listing 4.25 to Clojure generator source using Instacheck

Chapter 5

HTML and CSS Grammars (html5-css3-ebnf)

The **html5-css3-ebnf** module takes HTML5 and CSS3 specification data and translates it into EBNF grammar definitions that can be used for both generating and parsing web pages. The process is divided into three steps: parsing and pruning of specification data; translating the data to EBNF grammars; and assembling the translated grammars with boilerplate and low-level data definitions to create the final EBNF grammars. These steps are performed for both HTML5 and CSS3 data. This chapter describes BNF and EBNF grammars (5.1); the process of translating HTML specification data into EBNF grammars 5.2; the process of translating CSS specification data into EBNF grammars 5.3; and challenges that are presented when using those EBNF grammars for parsing and how those challenges are addressed in **html5-css3-ebnf** (5.4).

5.1 BNF and EBNF Background

BNF is a formal notation for describing context-free grammars. A BNF grammar is composed of 1 or more rules. Each rule contains a non-terminal symbol and an expansion separated by a "::=". Non-terminals are names surrounded by angle brackets. The expan-

```
<expr>    ::= <term> "+" <expr>
           |   <term>
<term>    ::= <factor> "*" <term>
           |   <factor>
<factor> ::= "(" <expr> ")"
           |   <integer>
```

Listing 5.1: Simple Math Expression BNF Grammar

```
expr   = term ( "+" expr )?
term   = factor ( "*" term )?
factor = "(" expr ")"   |  integer
```

Listing 5.2: Simple Math Expression EBNF Grammar

sion may contain one or more non-terminal symbols (referencing other expansion rules) and terminals (literals that should appear at that position). A terminal can be either a literal string enclosed in double-quotes or a terminal class (a unquoted name that is not surrounded by angle brackets) representing a class of literals (such as integers). A *juxtaposition* is a sequence of terminals and non-terminals in an expansion is indicates that every element must appear in the given order. In addition to *juxtaposition*, BNF expansion also supports *alternation* (a choice between multiple options) using a "|" symbol. Listing 5.1 depicts a BNF grammar for representing mathematical containing addition, multiplication and grouping.

Extended Backus-Naur form (EBNF) is the name for a number of variants of the original BNF that have a less verbose syntax with greater expressive power especially when dealing with repetition. Many EBNF variants also have a syntax for including regular expressions in the grammar definition. Our system uses the *Instaparse* library [14] for parsing EBNF grammars. Table 3.1 contains a summary of the EBNF syntax elements supported by *Instaparse*. In EBNF rules, the non-terminals can be specified without angle brackets and the symbol "=" can used instead of "::=" for rule definition. Listing 5.2 shows the simple mathematical BNF grammar from Listing 5.1 reexpressed in EBNF syntax.

5.2 HTML5 Grammar

5.2.1 HTML5 Background

The HTML language was originally defined by CERN [37] in 1990. In 1995 stewardship of the HTML definition was transferred to the newly created W3C organization. In 1998 the W3C stopped development on HTML and spent several years working on an Extensible Markup Language (XML) based reformulation of HTML known as Extensible HyperText Markup Language (XHTML). In 2004 Apple, Mozilla, and Opera created a new organization, Web Hypertext Application Technology Working Group (WHATWG) [38], to reinvigorate the development of HTML. The emphasis of the group was on backwards compatibility, changing the specification to match common implementations rather than trying to change implementations to match specifications, and writing specifications in enough detail to achieve complete interoperability between implementations. WHATWG release a new a version of the HTML standard called HTML5. In 2007 the W3C and WHATWG organizations began to work together on the HTML standard. The W3C now manages the development and maturing of the official HTML5 specification while the WHATWG organization continues to maintain a version of the HTML specification that includes features that are more exploratory in nature and that may eventually be standardized by the W3C [39].

5.2.2 Parsing and Translating HTML5

In addition to the human readable official standard specification published by the W3C [2], the W3C also maintains a structured version of the specification data [40]. This latter version is the data source that is used by the **html5-css3-ebnf** module. While this data is more structured than the readable format, it is still in HTML tables that are more convenient for assembly into readable specification documents than for machine processing. For this reason we still must parse this specification data into a data structure that we can use for translation.

```
 <tr>
  <th> Element
  </th><th> Description
  </th><th> Categories
  </th><th> Parents†
  </th><th> Children
  </th><th> Attributes
  </th><th> Interface
</th></tr>
```

```
<tr>
 <th><{a}></th>
 <td>Hyperlink</td>
 <td><a lt="Flow content">flow</a>;
     <a lt="Phrasing content">phrasing</a>*;
     <a lt="Interactive content">interactive</a></td>
 <td><a lt="Phrasing content">phrasing</a></td>
 <td><a>transparent</a>*</td>
 <td><a lt="global attributes">globals</a>;
     <{links/href}>;
     <{links/target}>;
     <{links/download}>;
     <{links/rel}>;
     <{links/hreflang}>;
     <{links/type}></td>
 <td>{{HTMLAnchorElement}}</td>
</tr>
```

Listing 5.3: W3C HTML Elements Table Header and <a>Entry

```
<tr>
  <th>Attribute</th>
  <th>Element(s)</th>
  <th>Description</th>
  <th>Value</th>
</tr>
```

```
<tr>
  <th><code>href</code></th>
  <td><{a}>; <{area}></td>
  <td>Address of the <a>hyperlink</a></td>
  <td><a>Valid URL potentially surrounded by spaces</a></td>
</tr>
```

Listing 5.4: W3C HTML Attributes Table Header and <href>Entry

```
{:element       "a",
 :description   "Hyperlink",
 :categories    ["flow" "phrasing*" "interactive"],
 :parents       "phrasing",
 :children      "transparent*",
 :attributes    ("links/href"
                 "links/target"
                 "links/download"
                 "links/rel"
                 "links/hreflang"
                 "links/type"),
 :interface     "HTMLAnchorElement"}
```

Listing 5.5: Parsed data for W3C HTML Element “<a>”

```
{:attribute    "a/href",
 :description "Address of the hyperlink",
 :value       {:raw "Valid URL potentially surrounded by spaces",
               :literals #{},
               :metas #{"Valid URL potentially surrounded by spaces"}},
 :element     "a"}
```

Listing 5.6: Parsed data for W3C HTML Attribute “href”

```
element = '<a' (<rS> a-attribute)* '>' (element | content)* '</a>'

a-attribute = global-attribute
            | 'download="' attr-val-a__download '"'
            | 'href="' attr-val-a__href '"'
            | 'hreflang="' attr-val-a__hreflang '"'
            | 'rel="' attr-val-a__rel '"'
            | 'rev="' attr-val-a__rev '"'
            | 'target="' attr-val-a__target '"'
            | 'type="' attr-val-a__type '"'

attr-val-a__href = ''
                 | url

url = #'[A-Za-z0-9$_@.,&+%=;/#?:-]*[A-Za-z0-9$_@.&+%=;/#?:-]'
```

Listing 5.7: Final EBNF Generated for the W3C HTML Element “<a>”

There are four main sources of data that are parsed by **html5-css3-ebnf** to extract definitions: the elements definition table, the attributes definition table, and two tables that define the form input elements. The top of listing 5.3 shows the header fields for the table that defines HTML elements. The bottom of listing 5.3 shows an example of the definition of the `<a>` element from that table. The top of listing 5.4 shows the header fields for the table that defines HTML attributes. The bottom of listing 5.4 shows an example of the definition of the `href` attribute from that table.

The **html5-css3-ebnf** module uses the Clojure *hickory* [41] library to parse the data from each specification table. The header data for a table is zipped together with rest of the table row data in order to construct a sequence of associative maps for each table. Listing 5.5 shows the resulting Clojure data structure for the `<a>` element. and Listing 5.6 shows the result for the `href` attribute that applies to it.

The final step is to translate the parsed data into the equivalent EBNF representation. The attribute data is more challenging to translate than the element data due to the inclusion of a free form *Value* string that describes that valid values that the attribute may be set to. We did a survey of the strings that are used in this slot. The majority of the values occur multiple times and have a clear meaning such as "Valid integer" and "Valid MIME type". However, other values are more ambiguous such as "Varies*" or "Text". We settled on a lookup table for the obvious cases with a fallback that generates a generic attribute value string.

Listing 5.7 shows a portion of the final HTML EBNF grammar for the `<a>` element and the `href` attribute. The "url" non-terminal for the `href` attribute is defined in other specifications [42] and not within the the W3C HTML standard itself. There are number of elements that fall into this category and we define these manually in a separate common EBNF definition file that we merge into the final EBNF grammar file.

```html
<!DOCTYPE html>
<html>
  <head>
    <style>
      div {
        color: red
      }
      .my-class {
        color: green
      }
      #my-id {
        color: blue;
        font-style: italic
      }
    </style>
  </head>
  <body>
    <div>
      Red text
    </div>
    <div class="my-class">
      Green text
    </div>
    <div id="my-id">
      Italic blue text
    </div>
  </body>
</html>
```

Listing 5.8: HTML file with CSS defined using an embedded “<style>” tag

5.3 CSS3 Grammar

5.3.1 CSS3 Background

The standardization of the CSS language is overseen by the the W3C [3]. The current version of the CSS language as a whole is “Level 3”. Previous versions of the CSS standard (Levels 1, 2, and 2.1) were defined as a single monolithic standard. Cascading Style Sheet Level 3 (CSS3) is composed of individual modules that are developed in parallel and can have their own level designation (some are currently at Level 4). As of Nov 2019, 40 CSS modules at or nearing “candidate recommendation” maturity or higher and there are more

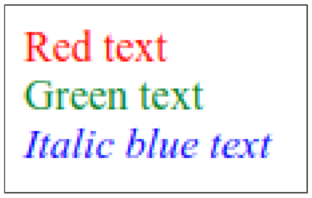
```
div {
  color: red
}
.my-class {
  color: green
}
#my-id {
  color: blue;
  font-style: italic
}
```

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="colors-style-file.css">
  </head>
  <body>
    <div>
      Red text
    </div>
    <div class="my-class">
      Green text
    </div>
    <div id="my-id">
      Italic blue text
    </div>
  </body>
</html>
```

Listing 5.9: CSS defined in a separate file and included into the HTML

```html
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <div style="color: red">
      Red text
    </div>
    <div style="color: green">
      Green text
    </div>
    <div style="color: blue; font-style: italic">
      Italic blue text
    </div>
  </body>
</html>
```

Listing 5.10: HTML file with CSS embedded inline within “style” attributes of individual elements



Red text
Green text
Italic blue text

Listing 5.11: Rendering of the the HTML and CSS from listings 5.8, 5.9, and 5.10

than 50 additional modules that are in a working draft state [43].

CSS is a domain-specific language or grammar that is used to modify the appearance or behavior of HTML elements on a web page. CSS is composed of two core components: properties and selectors. Groups of properties are delineated by curly braces. Within a group multiple properties are delimited by a semi-colon. Each property is composed of a property name followed by a colon followed by one or more property values. Property values are the most complex part of CSS and formal grammar used to define them is described in section 5.3.2.

CSS elements can be specified in three different contexts: in a separate file containing only CSS definitions (listing 5.9), as the content of HTML *style* elements (5.8), or contained within the value of *style* attributes (5.10). The first two forms of CSS (separate file and style element) have CSS selectors that prefix each CSS property group. The selectors indicate the HTML elements that the following properties apply to. The third CSS form (element style attribute) applies to the element the attribute is attached to, so no selector is needed (or permitted). Listings 5.8 - 5.10 show the different contexts where CSS can be specified. The result is the same in all three cases and is shown in Listing 5.11.

5.3.2 The CSS3 VDS Grammar

CSS property values in the CSS3 specifications are defined using a domain-specific formal grammar called VDS [16] [44]. As noted in 2.5.2, the VDS grammar is similar to EBNF with some additional features that make it more convenient for expressing CSS property values [45]. The grammar of VDS is defined in listings 5.12 and 5.13. Note that the VDS grammar is itself defined in terms of EBNF itself. In addition to the EBNF syntax elements listed in Table 3.1, the middle section of Table C.1 in Appendix C lists some of the additional EBNF that are used to define the VDS grammar.

```
assignments = assignment (<rS> assignment)* <S>?
assignment  = non-terminal <'='> <S> single-bar

<non-terminal> = <S> non-property <S>
               | <S> property     <S>

(* Operator precedence: mult, juxt, &&, ||, | *)
single-bar = double-bar (<rS> <'|'> <rS> double-bar)*
double-bar = double-amp (<rS> <'||'> <rS> double-amp)*
double-amp = juxtapose (<rS> <'&&'> <rS> juxtapose)*
juxtapose  = (<S> component ( comma? | <rS> ) )* component

component              = component-single
                       | component-multiplied
component-single       = literal
                       / func
                       / keyword-value
                       / non-property
                       / property
                       / brackets
                       / block
component-multiplied = component-single multiplier
multiplier             = question
                       | asterisk
                       | plus
                       | braces
                       | hash
                       | hash-braces
```

Listing 5.12: EBNF definition of the CSS VDS grammar (part 1)

```
keyword-value = identifier
non-property  = <'<'> identifier <'>'>
property      = <"<'"> identifier <"'>">
brackets      = <'['> <S> single-bar <S> <']'> bang?
block         = '{' <S> single-bar <S> '}'
              | single-bar ';'
func          = identifier <'('> <S> single-bar <S> <')'>
hash-braces   = <hash> braces
question      = <'?'>
asterisk      = <'*'>
plus          = <'+'>
braces        = bracesA
              | bracesA-B
              | bracesA-
bracesA       = <'{'> digit+ <'}'>
bracesA-B     = <'{'> digit+ <','> digit+ <'}'>
bracesA-      = <'{'> digit+ <',}'>
hash          = <'#'>
bang          = <'!'>

literal = #'[-_a-zA-Z0-9,/:;%]+'
        | '"'
        | <"'"> #"[-_a-zA-Z0-9,~|^$.<>={}:
 ↪  \u0028\u0029/\u005B\u005D\u002A\u002B]*" <"'">
        | #'[\u0028\u0029]'

<identifier> = #'@?[-_a-zA-Z][-a-zA-Z0-9]*(?:\u0028\u0029)?(?x)
 ↪  #identifier'
digit        = #'[0-9]'
comma        = ','
S            = #'\s*'
rS           = #'\s+'
```

Listing 5.13: EBNF definition of the CSS VDS grammar (part 2)

5.3.3 Parsing and Translating CSS3

The W3C publishes specifications for CSS modules but this information is spread across many separate documents and is designed primarily for human consumption so it is not an especially convenient format for programmatic access. However, the MDN maintains a repository [46] that aggregates the CSS definitions from those modules into comprehensive JSON data files.

There are three main JSON files that are loaded by **html5-css3-ebnf** to extract CSS data: property definitions, syntax definitions, and at-rule definitions.

The property definition file contains an dictionary of CSS properties with a mapping of property name keys to property value definitions. The property values themselves are a dictionary of attributes that define the property. Listing 5.1 shows the JSON definition of the "border" property. The most relevant field for both parsing of and generating properties is the "syntax" field which contains the VDS grammar that defines the valid "border" values.

The syntax definition file contains additional VDS definitions that is common to multiple properties. For example, the syntax of the "border" property contains a reference to "<line-width>". Listing 5.2 shows the definition of "<line-width>" from the syntax definition file. There are 10 other properties that refer to "<line-width>". However, note that "<line-width>" is not itself a top-level CSS property. The "<line-width>" syntax also refers to "<length>". Types are defined in a types definition file. Listing 5.4 shows the types file definition of the "<length>" type. Note that the types files does not provide any syntactical definition but just a link to a human readable web page that defines the type. In **html5-css3-ebnf** we used the human readable definitions to manually define the types in a separate EBNF grammar file that is merged into the final EBNF grammar definition.

The at-rules definition file defines a set of CSS statements that define some CSS behaviors. The generators of our system do not currently generate at-rules but many web pages use them so it is important to support them to be able to parse existing pages. Listing 5.3 shows the definition of the the "@media" at-rule. This rule allows groups of CSS properties to

```
{
 ...
 "border": {
   "syntax": "<line-width> || <line-style> || <color>",
   "media": "visual",
   "inherited": false,
   "animationType": [
     "border-color",
     "border-style",
     "border-width"
   ],
   "percentages": "no",
   "groups": [
     "CSS Backgrounds and Borders"
   ],
   "initial": [
     "border-width",
     "border-style",
     "border-color"
   ],
   "appliesto": "allElements",
   "computed": [
     "border-width",
     "border-style",
     "border-color"
   ],
   "order": "orderOfAppearance",
   "alsoAppliesTo": [
     "::first-letter"
   ],
   "status": "standard",
   "mdn_url": "https://developer.mozilla.org/docs/Web/CSS/border"
 },
 ...
}
```

Figure 5.1: Definition of the “border” property in MDN CSS repository

```
{
 ...
 "line-width": {
   "syntax": "<length> | thin | medium | thick"
 },
 ...
}
```

Figure 5.2: Definition of the “line-width” syntax in MDN CSS repository

```
{
 ...
 "@media": {
   "syntax": "@media <media-query-list> {\n  <group-rule-body>\n}",
   "interfaces": [
     "CSSGroupingRule",
     "CSSConditionRule",
     "CSSMediaRule",
     "CSSCustomMediaRule"
   ],
   "groups": [
     "CSS Conditional Rules",
     "Media Queries"
   ],
   "status": "standard",
   "mdn_url": "https://developer.mozilla.org/docs/Web/CSS/@media"
 },
 ...
}
```

Figure 5.3: Definition of the “@media” at-rule in MDN CSS repository

```
{
 ...
 "length": {
   "groups": [
     "CSS Types"
   ],
   "status": "standard",
   "mdn_url": "https://developer.mozilla.org/docs/Web/CSS/length"
 },
 ...
}
```

Figure 5.4: Definition of the “length” type in MDN CSS repository

```
prop-border =
  (
    nonprop-all |
    ( (
      nonprop-line-width |
      nonprop-line-style |
      nonprop-color
    )+ )
  ) ;

nonprop-line-width =
  (
    nonprop-length |
    ('thin' S) |
    ('medium' S) |
    ('thick' S)
  ) ;

nonprop-length      = any-number length-unit S
                    | '0' S ;
```

Figure 5.5: Translation of “border”, “line-width”, and “length” to EBNF

enabled or disabled depending on features of the media that the web page is being rendered for. The “@media” at-rule is commonly used to make adjustments for different screen sizes and for printing.

Because the MDN repository [46] uses the JSON format to define the CSS components, there is no special parsing needed to import that top-level data. However, the syntax field of every definition contains values defined using VDS grammar. This data is parsed by the VDS parser defined in Section 5.3.2.

The final step is translating the parsed VDS syntax data into the final EBNF grammar. This accomplished by performing a depth-first walk of the parsed syntax tree and translating each VDS combinator and terminal value into an equivalent EBNF representation. We keep track of the call depth of each generated element so that the final grammar is indented correctly which makes the EBNF easier to understand. This was especially important to validate the final EBNF grammar against the original VDS grammar. Listing 5.5 shows the final translated EBNF for the “border” property, “line-width” syntax, and “length” type.

5.4 Parsing Challenges

One of challenges with using the HTML and CSS specification derived grammars for parsing is that many web pages in the wild are not actually compliant with the W3C standards. In addition to obsolete elements that were never fully standardized, web pages often contain misspelled HTML elements/attributes and CSS property names/values.

Another challenge related to parsing is related to the use of Instaparse library for parsing and its support for ambiguous grammars. While this support provides significant expressive power when defining grammars, it can sometimes result in the need for Instaparse to do significant backtracking in order to find a path through the grammar that is able to parse the full text. Instaparse can return all possible ways to parse a text given an ambiguous grammar. However, by default, Instaparse will short circuit and return the first successful parse through the text. However, even with short circuiting, certain combinations of ambiguous grammars and texts can be inefficient. This is especially true if the web page is not valid and there are no valid parses that will satisfy the grammar. In that case, Instaparse will need to exhaustively search all optional paths through the grammar before determining that there are no possible valid parses.

To address these challenges, **html5-css3-ebnf** provides support for multiple parsing passes. The first pass uses a much simpler *tags and attributes* grammar with less ambiguity in the grammar. Listing D.1 of appendix contains the EBNF definition of this grammar. This grammar encodes the basic syntactic and structural characteristics of an HTML page with the specification-based constraints of the full grammar. In addition, the grammar accepts arbitrary tag names, attribute names and values and also parses tags with no closing pair.

Once the web page is parsed with the *tags and attributes* parser, several transformation are applied to the the resulting parse data to transform the web page before it is passed to the next parsing phases. These transforms include the following:

- Remove elements that can be large and complicated to parse but are not directly relevant for web page render testing such as the content of `<script>` and `<svg>` tags.

- Deduplicate attributes that occur multiple times in the HTML tag by either removing subsequent occurrences or by combining the content of the attributes into a single attribute.
- Remove extra whitespace that does not have any semantic or visual impact.
- Prune or rewrite tags, attributes, and attribute values that match certain patterns that are known to happen on real-world web pages that can pose problem for the later parsing phases. For example, many sites set the value of the table border attribute to "0" to hide the border. However, this is not valid according to the specification so the value is replaced with an empty string.
- Split apart CSS style information contained in `<style>` elements and in `style` attributes.

The *tags and attributes* parsing phase returns the HTML and CSS data as separate values which are then parsed in separate passes. This enables the grammars for HTML and CSS to be defined and used separately which reduces the complexity and ambiguity of the grammars resulting in faster and more memory efficient parsing.

The *tags and attributes* grammar can also be used as a standalone library without the subsequent parsing phases. This allows subsets of the the transformations listed above to be applied or omitted. There are also additional translations that are not applied during the normal parsing process but can be applied on demand. One feature of the *tags and attributes* parser is that it retains all whitespace information from the original web page. This allows narrow transformations to be made against the HTML data without visual changes to the whitespace and indenting. The parser also provides a mode that can replace the whitespace data of the page with properly indented whitespace.

5.5 html5-css3-ebnf Usage

Over time the HTML and CSS standards are changed as new components are added and current components are modified or improved. The **html5-css3-ebnf** project (available at https://github.com/kanaka/html5-css3-ebnf) can be used from the command line to regenerate the EBNF grammar files and cached parsing grammar files to incorporate those changes. The CSS regeneration mode also supports the ability to filter the selected components of the standard by their standardization maturity. In addition, the parse mode supports parsing of web pages to extract Instacheck style wtreks or to prune the EBNF grammar based on the parsed web page.

The examples below use the Leiningen project [36] for Clojure project management and execution. Leiningen is invoked via the `lein` command.

5.5.1 Generate EBNF Parsing Grammars

The primary function of the *html5-css3-ebnf* module is to translate W3C HTML and CSS specification data into EBNF grammars that can parse web page test cases that comply with those specifications. Listings 5.14 and 5.15 show the commands to generate the HTML and CSS EBNF grammars from the W3C specification data. The HTML and CSS specification data is contained in the `w3c_html` and `mdn_data` directories respectively. These directories

```
time lein with-profile html5 run
```

```
Generating HTML5 EBNF based on: (./w3c_html/sections/elements.include
↪  ./w3c_html/sections/attributes.include ./w3c_html/sections/semantics-forms.include
↪  ./resources/html5-prefix.ebnf ./resources/html5-base.ebnf ./resources/common.ebnf)
Checking EBNF and converting to Parser
Converting Parser to Cached Grammar
Saving HTML5 EBNF to ./data/html5.ebnf
Saving cached parser grammar EDN to ./data/html5.grammar
```

Listing 5.14: Generate/regenerate the HTML EBNF grammars from the W3C specification data (command and output)

```
time lein with-profile css3 run
```

```
Creating VDS grammar parser from: ./resources/css-vds.ebnf
Generating full CSS VDS grammar based on: ./mdn_data/css/properties.json
↪  mdn_data/css/syntaxes.json mdn_data/css/at-rules.json
Saving full CSS VDS grammar file to: ./data/css3.vds
Parsing CSS VDS grammar
Converting CSS VDS grammar to EBNF
Saving EBNF to ./data/css3.ebnf
Checking EBNF and converting to Parser
Converting Parser to Cached Grammar
Saving cached parser grammar EDN to ./data/css3.grammar
```

Listing 5.15: Generate/regenerate the CSS EBNF grammars from the W3C specification data (command and output)

are git submodules [47] that pull the original repositories directly into the *html5-css3-ebnf* repository.

Selecting and Filtering Specification Data

The HTML and CSS specification data is being continually updated by the W3C. Normal git fetch and branching commands can be used within the `w3c_html` and `mdn_data` submodule directories to update to newer or older versions of the specification data. In addition, the CSS data also contains standards status values that can be used to filter parts of the standard. By default, the "standard" status value is selected for translation however, the command in Listing 5.16 demonstrates the use of the `--status-list` option to include both standard and obsolete parts of the CSS standard in the translated EBNF grammars. Valid status values include: "standard", "obsolete", "nonstandard", and "experimental".

```
time lein with-profile css3 run --status-list standard,obsolete
```

Listing 5.16: Generate/regenerate the CSS EBNF grammars with status filtering

5.5.2 Parse Web Pages

Once EBNF grammars have been generated from the specification data (as described by the process in), these grammars can then be used to parse web pages in order to validate that the web page adheres to the HTML and CSS specifications and to extract a path frequency weight trek from the web pages. The first command shown in Listing 5.17 demonstrates how to do this. The second command shows additional options that use Instacheck to output HTML and CSS EBNF grammars that have been pruned of all unreachable paths based on the parsed path frequency trek. More than one web page can be specified on the command line, in which case, all the pages will be parsed and the resulting path frequencies weights will represent a summation of all the individual path frequency weights. The combined path frequency trek is saved to a file and is also used for the grammar pruning.

Web Page Parsing Example

This section shows the results of using *html5-css3-ebnf* to parse a very simple web page that is shown in Listing 5.18. The web page contains a single `<div>` element containing a single “x” character. The element has a single `color` CSS property attached to it to change the font color of contained text. Listing 5.19 shows the full path frequency trek that results from parsing the web page in Listing 5.18.

This path frequency trek can then be used to prune unreachable parts of the HTML and CSS grammar and save these pruned grammars to disk (as shown in the second command in Listing 5.17). In the path frequency trek (Listing 5.19) the only top-level HTML element path that appears is `[:element :alt 26]` which is the alternation for the `<div>` element. All other element path alternations have an implicit path frequency of zero. The result is that in the pruned grammar in Listing 5.20 the rule `element = "<div" ">" content * "</div>"` contains a `<div>` element and no other elements. Likewise the path `[:css-known-standard :alt 98]` refers to the `color` CSS property name and is the only property name that occurs in the path frequency trek. This

```
time lein run page-to-parse.html --weights-output weights.edn
```

```
time lein run page-to-parse.html --weights-output weights.edn
 ↪  --html-ebnf-output html.ebnf --css-ebnf-output css.ebnf
```

```
Loading HTML parser
Loading CSS parser
Processing: 'page-to-parse.html'
  - parsing HTML
  - HTML weights: 13/1750
  - parsing CSS
  - CSS weights: 24/3576
Weight count: 36
Generating pruned HTML EBNF
Saving pruned HTML EBNF to: 'html.ebnf'
Generating pruned CSS EBNF
Saving pruned CSS EBNF to: 'css.ebnf'
Saving weights to: 'weights.edn'
```

Listing 5.17: Execute html5-css3-ebnf *parse* command to parse a web page (commands and output)

```
<html>
<body>
  <div style="color: red">
    x
  </div>
</body>
</html>
```

Listing 5.18: Simple HTML page containing a div tag with red text styling

```
{[:S :star 0] 3,
 [:S :star nil] 7,
 [:body :cat 1 :star nil] 1,
 [:body :cat 3 :star 0 :alt 0] 1,
 [:body :cat 3 :star 0] 1,
 [:content :alt 0] 1,
 [:css-assignments :alt 1 :cat 2 :star nil] 1,
 [:css-assignments :alt 1 :cat 3 :star nil] 1,
 [:css-assignments :alt 1] 1,
 [:css-combinator :alt 3] 1,
 [:css-declaration :alt 0 :cat 0 :ord 0] 1,
 [:css-declaration :alt 0 :cat 1 :alt 0] 1,
 [:css-declaration :alt 0] 1,
 [:css-known-standard :alt 98] 1,
 [:css-ruleset :cat 1 :star nil] 1,
 [:css-selector :cat 1 :star 0 :opt 0] 1,
 [:css-selector :cat 1 :star 0] 1,
 [:css-simple-selector :alt 0 :cat 0 :opt 0 :alt 1] 1,
 [:css-simple-selector :alt 0 :cat 0 :opt 0] 1,
 [:css-simple-selector :alt 0 :cat 1 :star nil] 1,
 [:css-simple-selector :alt 0] 1,
 [:css-simple-selector :alt 1] 1,
 [:css-univ :cat 0 :opt nil] 1,
 [:element :alt 26 :cat 1 :star nil] 1,
 [:element :alt 26 :cat 3 :star 0 :alt 1] 1,
 [:element :alt 26 :cat 3 :star 0] 1,
 [:element :alt 26] 1,
 [:html :cat 0 :opt nil] 1,
 [:html :cat 1 :star nil] 1,
 [:html :cat 3 :star nil] 1,
 [:html :cat 6 :opt nil] 1,
 [:nonprop-color :alt 5] 1,
 [:nonprop-named-color :alt 121] 1,
 [:prop-color :alt 1] 1,
 [:stylesheet :cat 1 :star 0 :alt 0] 1,
 [:stylesheet :cat 1 :star 0] 1}
```

Listing 5.19: Resulting path frequency trek from parsing HTML from Listing 5.18

```
html = "<html" ">" <S> body "</html>" <S>
body = "<body" ">" element* "</body>" <S>
element = "<div" ">" content* "</div>"
content = char-data
char-data = #"[^<&]*"
S = #"\s"*
```

Listing 5.20: Resulting HTML EBNF grammar after pruning the grammar using the path frequency trek in Listing 5.19

```
stylesheet = S css-ruleset*
css-ruleset = css-selector "{" S css-assignments "}" S
css-assignments = css-declaration S
css-declaration = css-known-standard
css-selector = css-simple-selector
               (css-combinator css-simple-selector)?*
css-combinator = rS
rS = #"\s+"
css-simple-selector = css-univ?
css-univ = "*"
css-known-standard = "color" S ":" S prop-color
prop-color = nonprop-color
nonprop-color = nonprop-named-color
nonprop-named-color = "red" S
S = #"\s"*
```

Listing 5.21: Resulting CSS EBNF grammar after pruning the grammar using the path frequency trek in Listing 5.19

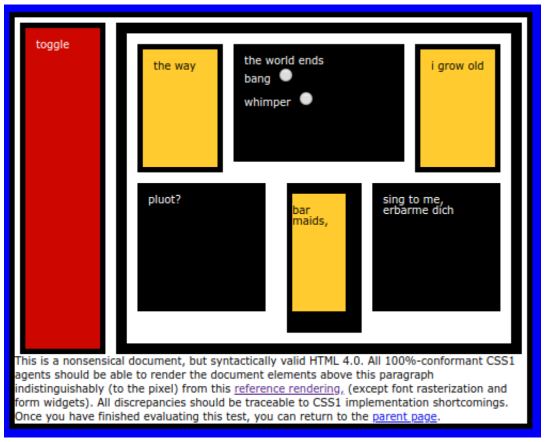
results in the pruned rule `css-known-standard = "color" S ":" S prop-color` in the grammar depicted in Listing 5.21.

Chapter 6

Bartender: Browser Automated Render TestiNg DrivER

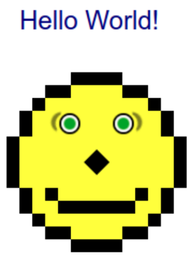
6.1 The State of Browser Render Testing

In 1998 the Web Standards Project (WaSP) [48] was formed to promote the use of web standards within browser implementations and use of those standard by the web developer community [49]. As part of the effort three *Acid* tests were developed to test web browser standards conformance. Acid1 [50] was released in 1998 [51] and incorporated much of the CSS 1.0 specification into a single web page that could be easily compared to a reference image. Figure 6.1 shows a rendering of the Acid1 test case. Acid2 [52] was released in 2005 and designed to test browser compliance with the CSS 2.1 specification [53]. Figure 6.2 shows a rendering of the Acid2 test case. Acid3 [54] was release by WaSP in 2008 and designed to test browser compliance with: the document object model (DOM) Level 2 specification, the first ECMAScript (JavaScript) standard, the Scalable Vector Graphics (SVG) specification, and some parts of the CSS Level 3 specification [55]. Acid3 uses a number of elements from those specifications that were still being standardized and it has diverged from the standards that were eventually adopted by browser vendors. Figure 6.3 shows a rendering of

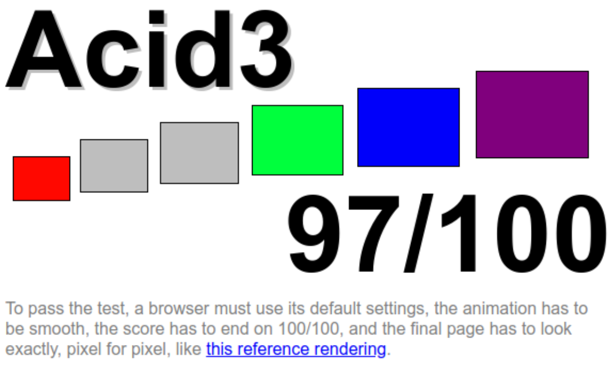


This is a nonsensical document, but syntactically valid HTML 4.0. All 100%-conformant CSS1 agents should be able to render the document elements above this paragraph indistinguishably (to the pixel) from this reference rendering, (except font rasterization and form widgets). All discrepancies should be traceable to CSS1 implementation shortcomings. Once you have finished evaluating this test, you can return to the parent page.

Listing 6.1: Rendering of Acid1 Test Case



Listing 6.2: Rendering of Acid2 Test Case



To pass the test, a browser must use its default settings, the animation has to be smooth, the score has to end on 100/100, and the final page has to look exactly, pixel for pixel, like this reference rendering.

Listing 6.3: Rendering of Acid3 Test Case

the Acid3 test in Chrome 73 indicating that only 97 out of 100 features tested by the page completed successfully. The WaSP group disbanded in 2013 after determining that their effort to promote web standards had been successful.

The web-platform-tests (WPT) [56] is a project that was launched by the W3C in 2014 as a consensus-driven (human consensus rather than automated testing consensus) open-source project that maintains a central repository of tests for the web platform [57]. There are three main types of tests that are part of the WPT project: reftests, visual tests, and manual tests. Reftests consist of two or more test cases that must render identically and can be checked automatically with assertions within the test case. Visual tests are test cases that are loaded and a screenshot is taken that is compared to an image with the correct rendering. Manual tests are test cases that require some sort of visual inspection or interaction in order to validate the final result [58].

Both WPT reftests and visual tests can be executed in an automated fashion, however there are still significant parts of the test life cycle that require manual human intervention. Reftests must be carefully crafted so that each page in the test results has assertions which check for the same rendering behavior [59]. The reference images that are used by visual tests are manually generated or manually validated [60].

We have developed a new approach called Bartender (**B**rowser **A**utomated **R**ender **T**esti**N**g **D**riv**ER**) that automates the entire testing process. It does this by using specification based generators to create random test cases and by using a consensus oracle of different browsers to determine an indirect measure of correctness for the resulting rendering. This measure is indirect and approximate because if all browsers in the consensus pool have the same defect then there will be consensus even though the behavior is incorrect. The Bartender approach is not a replacement but rather complements the WPT testing approach. It can detect edge cases that are tedious to cover comprehensively reftests. In addition, WPT test cases tend to focus on a single feature or element of a web specification. The Bartender approach can effectively cover the interactions between the large number of web specification features without the need for tests to be written manually that cover every interaction.

6.2 Bartender System Architecture

Figure 6.1 shows an complete depiction of the system architecture in introduced in Figure 2.1. The bottom of Figure 6.1 shows the process that is used to parse, cleanup, and translate HTML and CSS specification data into EBNF grammars. This happens in the html5-css3-ebnf module that is described in Chapter 5. Raw HTML specification data from the W3C [40] is parsed at 6.1-A into data structures describing the HTML element and attributes which is translated at 6.1-B into equivalent EBNF grammar data which is stored in the datastore at 6.1-E. Raw CSS VDS grammar is parsed from MDN [46] data to a VDS AST data structure at 6.1-C which is translated to equivalent EBNF grammars at 6.1-D and stored in the datastore (6.1-E). The EBNF grammars are then loaded, optimized for generating test cases, and translated into Clojure generator code 6.1-F. The process of optimizing and translating the grammars is described in more detail in Section 6.6.3.

The top of Figure 6.1 shows the runtime testing process of Bartender itself. When the test system is started, the runtime testing configuration (see G), the HTML and CSS EBNF grammars, and the HTML and CSS generators are loaded at 6.1-G. A web server is started at 6.1-H to serve static files from the datastore (6.1-E). WebDriver [13] connections are established with each browser in the browser consensus pool at 6.1-I.

The test harness then begins to execute iterations of the first Instacheck run. For each iteration, the generators are invoked to create an HTML and CSS test case which is stored in the datastore (6.1-E) and each browser is requested to load the test case via the WebDriver connection (6.1-I). Once the test case is rendered, the test driver again uses the WebDriver interface (6.1-I) to request that each browser take a screenshot of the rendered page and return it as a Portable Network Graphics (PNG) image to the test driver. Each screenshot is normalized, difference images are calculated for each browser pair, and thumbnail images

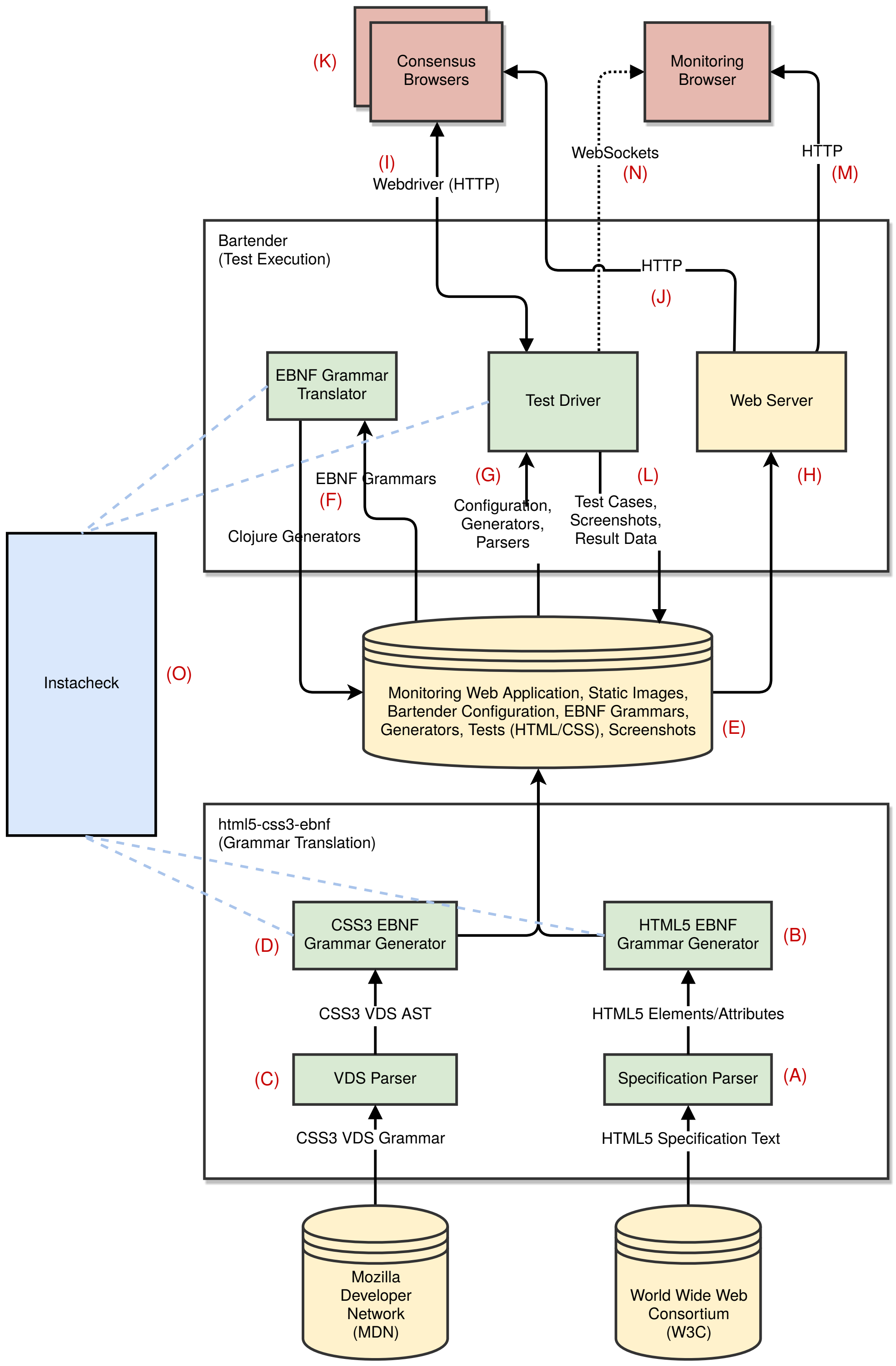


Figure 6.1: Bartender System Architecture

are generated. All this image data is stored at 6.1-L to the datastore (6.1-E).

Bartender also provides real-time monitoring of the testing process via a monitoring web application. The application is loaded from the datastore via the static files web server at 6.1-M. The application then makes a WebSocket connection back to the test driver to the current test driver state along with real-time updates as the testing process proceeds. The monitoring application is described in more detail in Section 6.5.

The core Bartender system is implemented in Clojure and the monitoring and reporting applications are written in ClojureScript [61] (a variant of Clojure that compiles to JavaScript). Clojure is very well suited to data processing and manipulation. In addition, the use of Clojure allowed us to use both Java libraries such as *OpenCV* and *Selenium WebDriver* as well as Clojure libraries such as *test.check*, *test.chuck*, *Instaparse*, *Hickory*, *specter*, *ring*, *differ*, and *transit*. The use of ClojureScript for the monitoring applications allows reuse of Clojure data structures and libraries that are used by the server. In addition, ClojureScript allows us to leverage JavaScript libraries like React. See Appendix I for the a description of the projects and libraries used in Bartender and Appendix J for code size and other statistics related to Bartender.

6.2.1 Optimizing and Translating EBNF Grammars to Clojure Generators

The *Instacheck* module provides the ability to translate EBNF grammars to Clojure generators at runtime. However, for large grammars like the HTML and CSS grammars this can take a long time and translation process requires substantially more memory than the using the translated grammars for test case generation. For this reason, Bartender uses Instacheck to pretranslate and cache the Clojure code as a separate one-time step prior to using them for testing (6.1-F). This process is described in more detail in Section 4.4. The Bartender command for executing this process is described in Section 6.6.4. The translated Clojure code is then loaded by Bartender via the normal Clojure code loading process.

```
S          = #'\s'* ;
rS         = #'\s+' ;
```

Listing 6.4: Whitespace EBNF Rules

The EBNF grammars that are generated by the *html5-css3-ebnf* module could be translated directly to Clojure generators code but there a number of cases where it is suboptimal to use the direct translation of EBNF grammars. During translation there are several categories of grammar modifications and optimizations that are applied to the grammar. A non-exhaustive list of some of the most common optimization categories are described below:

Whitespace

The grammar rules for white space matching are used throughout both the HTML and CSS grammar definitions. Listing 6.4 shows how these rules are defined. The *S* rule means that zero or more whitespace characters must appear at this location and the *rS* rule means one or more whitespace characters must appear at this location. A naive translation of these rules to generator functions would result in generators that would return sequences of spaces with a length up to the current PBT size limit for the generator. However, in both HTML and CSS there is no semantic difference between a single space and two or more adjacent spaces (unless they are literal quoted string values). For example, Listing 6.5 shows two different portions of HTML that would be rendered identically by the browser. For this reason, during the optimization phase, whitespace rules in the grammar are replaced with a simple generator that returns a single space character.

Scalar Types

One of the limitations with the use of EBNF as the fundamental way to define the generators is that there is no predefined rules for fundamental scalar types and so the scalar

```
<div> This is a test </div>
```

```
<div>  This   is          a
  test
</div>
```

Listing 6.5: Repeated Whitespace is Ignored in HTML

```
integer = #"-?[0-9]+" ;
non-negative-integer = #"[0-9]+" ;
positive-integer = #"[1-9][0-9]*" ;
floating-point-number = #"-?[0-9]*[.][0-9]+(?:[eE]-?[0-9]+)?" ;
any-number = integer | floating-point-number ;
```

Listing 6.6: Whitespace EBNF Rules

types are defined using string and regular expression primitives. Listing 6.6 shows the definition for numeric types in the EBNF grammars. These definitions are acceptable for parsing but there are problems with using the definitions directly for generation. The first problem is that generators derived from these definitions are less efficient than the native scalar generators provided by the *test.check* library. A more significant problem is that these rules do not take advantage of the type specific sizing and shrinking capabilities of the *test.check* scalar generators. For example, a generator based directly on the *floating-point-number* EBNF rule will generate a scientific notation about 50% of the time and does not consider 1.000 to be a simpler value than 5.789 for the purposes of sizing and shrinking. For this reason, during the optimization phase, rules for numeric types are replaced with the equivalent *test.check* generators.

Image URLs

The EBNF definition for URLs makes sure that those URLs are well formed. However, in actually test cases, these URLs have a context dependent meaning. For example, the *href* attribute of an *img* tag is used to load an image from that URL to be rendered on the page. In particular, images are an important use case that we want to test. A naive translation of the EBNF grammar will result in URLs that are effectively random strings that have a near zero change of referring to images being served by the embedded webserver. For this reason, during the optimization, phase the generator for image URLs is replaced with generator that returns image URLs that point to a limited set of images that are actually being served by the webserver.

Bug Workarounds

The optimization phase can also be used to fix certain classes of browser bugs that may interfere with determining consensus. One of the Servo bugs that was discovered during testing is that adjacent characters in the HTML that use the Ahem font have a faint color border between the characters that can be reported as a rendering difference (`https://github.com/servo/servo/issues/24042`). To workaround this issue we replace the character data rule with one that wraps each character with a special span tag that sets the background color of the span with the color of the font itself.

6.2.2 Bartender Testing Process

The following process happens when Bartender starts up:

1. The user configuration is loaded and parsed (6.1-E).

2. Any additional wtrek (weight) files are loaded and merged into the active wtrek configuration (6.1-E).

3. The embedded web server is started (6.1-H). The WebSockets endpoint (6.1-H) is also started which is used to update monitoring applications as the test state changes (6.1-N).

4. The cache parser tree data structures for both HTML and CSS are loaded (6.1-G). These are used to parse wtrek data from generated test cases.

5. A WebDriver session is established with each browser that is configured in the consensus pool (6.1-I).

6. The starting configuration and runtime state is sent to any connected monitoring applications (6.1-N).

The following steps occur during each Bartender test run:

1. The run number and current random seed are updated.

2. The test case generator is updated with the current weight configuration (wtrek).

3. Instacheck is used to execute the quick-check iterations using the test case generator and check function which serves as both the test executor and test oracle.

4. If weight reduction is configured and the quick-check process detected a counterexample/failure then Instacheck is used to adjust the active wtrek weights with the configured reduction algorithms.

5. The current wtrek (weight) configuration is stored in the datastore (6.1-L).

The following steps occur during each Bartender PBT test iteration:

1. A new web page is generated from the HTML and CSS grammar generators and stored in the datastore (6.1-L).

2. The web page is loaded by each browser in the consensus pool (6.1-I and 6.1-J).

3. A screenshot of the current rendered state is taken (6.1-I) and stored in the datastore (6.1-L) for each browser in the consensus pool .

4. Difference images are created and stored in the datastore for each pair of browser screenshots (6.1-L).

5. Thumbnail images are created and stored in the datastore for all screenshots and difference images (6.1-L).

6. The configured comparison algorithm is applied to generate a disagreement measure which is compared to configured threshold value to determine if the screenshots are different enough to be added to the set of violations.

7. The configured consensus algorithm is then applied to the set of violations from the previous step to determine if test case should be considered an overall failure/counterexample. If so, then subsequent iterations are part of the shrinking phase. See more about consensus in Section 6.3.

8. During the shrinking phase, any time a smaller failure is discovered, the test case is parsed using Instacheck to extract its path frequency weight trek and a summary of the tags, attributes, and properties on the page. This path frequency weight trek is used as an input to weight reduction process between test runs.

9. The run state is updated with the results of the test iteration. This triggers a delta packet to be sent to all connected monitoring applications (6.1-N).

6.3 Consensus

The HTML and CSS specifications are both quite large and undergoing frequent updates. A test oracle that can automatically and comprehensively perform formal verification of web browser rendering behavior approaches the complexity of developing a web browser rendering itself. In addition, this form of test oracle is likely to have its own defects that will reduce its utility for formal verification of other SUTs. For this reason most browser render testing involves manual human involvement at various stages of the testing process. The approach taken with Bartender is to use multiple browsers as a consensus test oracle (this was introduced in Section 2.7). This is a form of differential testing where multiple SUTs are compared and differences in behavior indicate that the test case is a counterexample candidate [19].

A lack of consensus may indicate a defect exists in one of the SUTs, but in some cases the behavior differences may be permissible and not indicate a defect. In addition, the presence of consensus also does not necessarily imply that a test case is defect free. It is possible that all implementations in a consensus pool have a defect which affects behavior in the same way. However, in the context of web browser rendering engines, common behavior may be a more important measure of quality than a low number of defects. An alternate way of expressing this is that a "difference in behavior" can itself be considered a defect in this context. In fact, a less widely used browser that has better adherence to a formally defined specification may be considered the "defective" browser in comparison to the more widely used browser. The situation where a SUT prioritizes common behavior over strict formal correctness is known as "bug compatibility" [62]. The desire for "bug compatibility" is another reason that a traditional test oracle is not optimal for browser render testing. However, the consensus oracle approach used by Bartender, by definition, detects differences between browsers via lack of consensus.

6.3.1 Consensus Algorithms

There are two components of determining browser consensus for any given PBT test iteration (a single page load/render): calculate a disagreement measure between each pair of browsers and determining whether the set of disagreement measure represent a consensus or represent a lack of consensus and therefore a test failure/counterexample. The default method for calculating the disagreement measure value is to perform a normalized SSD between the color channels of each pair of rendered browser screenshot images and taking the mean across the color channels. The default method for deciding on consensus from a set of disagreement

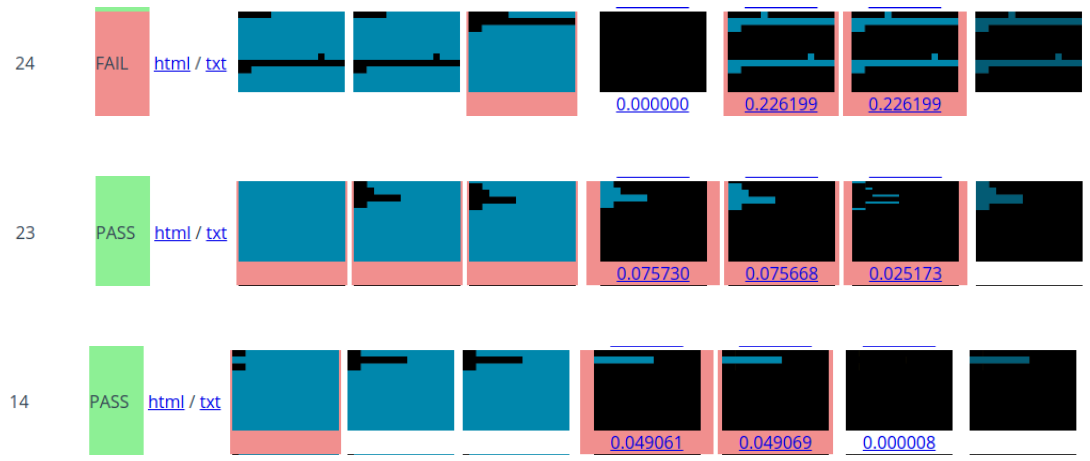


Figure 6.2: Test iteration example showing: Servo with a rendering that different from Firefox and Chrome; Servo, Firefox, and Chrome all with different renderings; Firefox with a rendering that is different (Servo and Chrome are in agreement)

measures is to compare each measure to a threshold value. If all disagreement values are under the threshold value then the iteration is considered to have consensus, otherwise it is considered to be a failing case.

The default methods of calculating disagreement measures and overall consensus are described in more detail in Section 2.7.2. However, these methods are not the only options for determining consensus. The most useful consensus method is often highly dependent on the specific goals of the testing being performed. The Bartender system has several other consensus algorithms that are available and the system has been designed to allow other consensus algorithms to be developed and integrated. For the disagreement measure Bartender uses the OpenCV library [63] and any of the OpenCV template matching functions are available for consensus calculation include the normalized and unnormalized versions of SQDIFF, CCORR, and CCOEFF [64]. In addition, the threshold that the results are compared against is user configurable. This can be used to adjust the sensitivity of the consensus to small variations in the rendering screenshots.

The default overall Bartender consensus algorithm is to treat any difference measure violating the threshold as a lack of overall consensus. Bartender also supports a “target” mode in which one browser in the pool is the focus of the testing and the other browsers in the pool are used as a test oracle to for the targeted browser.

Figure 6.2 shows some example iterations from the monitoring application during targeted testing of Servo. In each image the columns represent the following: iteration number, consensus result, Firefox, Chrome, and Servo screenshots, Firefox/Chrome, Chrome/Servo, and Servo/Firefox difference images, and an average of the browser screenshots (the monitoring application is described in more detail in Section 6.5).

The only test cases that are considered failures are when the Servo rendering for the test case is different from both Firefox and Chrome but Firefox and Chrome do not show a difference. This is the case shown in the first image of Figure 6.2. If all three browsers have a different rendering then the test case is not considered a failing case (counterexample). In the targeted testing configuration, the common behavior of Firefox and Chrome is considered to be the test oracle for the targeted browser, Servo. When Firefox and Chrome disagree then there is not enough information about the behavior of Servo to determine its correctness so the overall test is considered passing. This is the case shown in the second image of Figure 6.2. Another case that is considered to be passing is if Servo's rendering behavior is the same as Firefox and different from Chrome or vice versa. This is another case where ambiguity exists in the Firefox+Chrome test oracle so the overall test case is considered passing. The third image of Figure 6.2 depicts the case where Servo is different from Firefox but has the same rendering as Chrome.

The targeted testing algorithm is described above in terms of three browsers, but the algorithm applies to any number of browsers (greater than one). The test case is considered failing (a counterexample) if, and only if, the targeted browser is in disagreement with all other browsers, and none of non-targeted browsers are in disagreement with each other. Disagreement in this case means the disagreement measure violates the threshold. Note that in a two browser configuration (one target, one non-target) the targeted testing mode is equivalent to the default overall consensus algorithm in which any disagreement is considered a failing test case.

6.3.2 Consensus Challenges and Solutions

The release of the HTML5 proposed specification by the WHATWG organization was an attempt recognize and formalize the areas of HTML where the dominant browsers already had common behavior and to either deprecate or tighten up areas with divergent behavior (see Section 5.2.1). This has reduced the need for "bug compatibility" between browser implementations. In addition most browsers support "Quirks Mode" rendering which is a backwards compatibility mode for web pages that assume certain browser behavior that are no longer considered desirable [65] [66]. The current test cases generated by Bartender do not trigger "Quirks Mode". However, there are still a number of challenges that must be addressed to make a consensus oracle useful within a browser render testing context. Section 2.7.1 gave a brief introduction to challenges that exist in using multiple browser rendering engines as a consensus test oracle.

The key challenge, is that in the context of web browser rendering there a differences in behavior that may be considered permissible (false positives). There are a number of reasons for differences in behavior including:

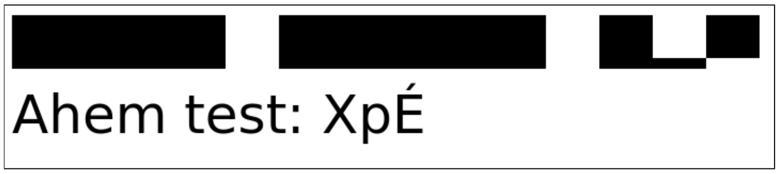
- Configuration differences between the browsers.
- Host operating system differences that may affect each browsers in different ways.
- Proprietary features that are only supported by a subset of browsers.
- Obsolete features that have been removed or have different behavior in a subset of the browsers.
- Experimental features that are present or have different behavior in a subset of the browsers.
- Features that have standard specifications but which are poorly, or under specified, and are interpreted differently by a subset of the browsers.

- Unintended defects in the implementation of features that exist in a subset of the browsers.

A simple solution to permissible rendering differences is to simply configure grammar weights to avoid these cases. However, this would eliminate a large number of important HTML and CSS elements that are desirable to test if possible. For example, font rendering is a particularly problematic area where browsers vary widely in rendering behavior, and yet, nearly every real-world web page contains text, so omitting fonts from testing would be a significant testing gap. In addition, text on a web page may interact with many other elements on a page ways that may not be testable by other means. Testing font rendering is challenging, but it is also important to include as part of comprehensive browser render testing. Other browser rendering differences occur due to the varying browser defaults for many CSS styling properties. Many of these setting variations are in common page elements so omitting them is not a good option. For example, even the default margin between the browser control elements and the content of the page varies among browsers. Solutions to the these consensus challenges are discussed below.

The Ahem Font

Including text in web page test cases increases the likelihood of revealing rendering defects in a browser. Omitting fonts from testing would be a significant testing gap. However, fonts have a wide variation in the way that they are rendered on a web page and these differences do not necessarily indicate a defect. Browsers attempt to select the best match between the font requested by the current CSS style setting and the fonts that are available to the



Listing 6.7: Example of "Ahem test: Xp&#x00c9;" string rendered with Ahem font

```
<html>
    <head>
    <style>
        @font-face {
            font-family: Ahem;
            src: url(Ahem.ttf);
        }
        body {
            font: 50px/1 Ahem;
            color: #808000;
        }
    </style>
    </head>
    <body>
        abc<br>
        cd &nbsp; e<br>
    </body>
</html>
```

Listing 6.8: Web page demonstrating Servo Ahem font bug 24042

```
.wrap-ahem {
    font: 25px/1 Ahem;
    margin: 0px;
    background: currentColor; /* match font color */
}
```

```
<span class="wrap-ahem">X</span>
```

Listing 6.9: Workaround Servo Ahem render bug 24042 by wrapping each character in a span with color matched background color

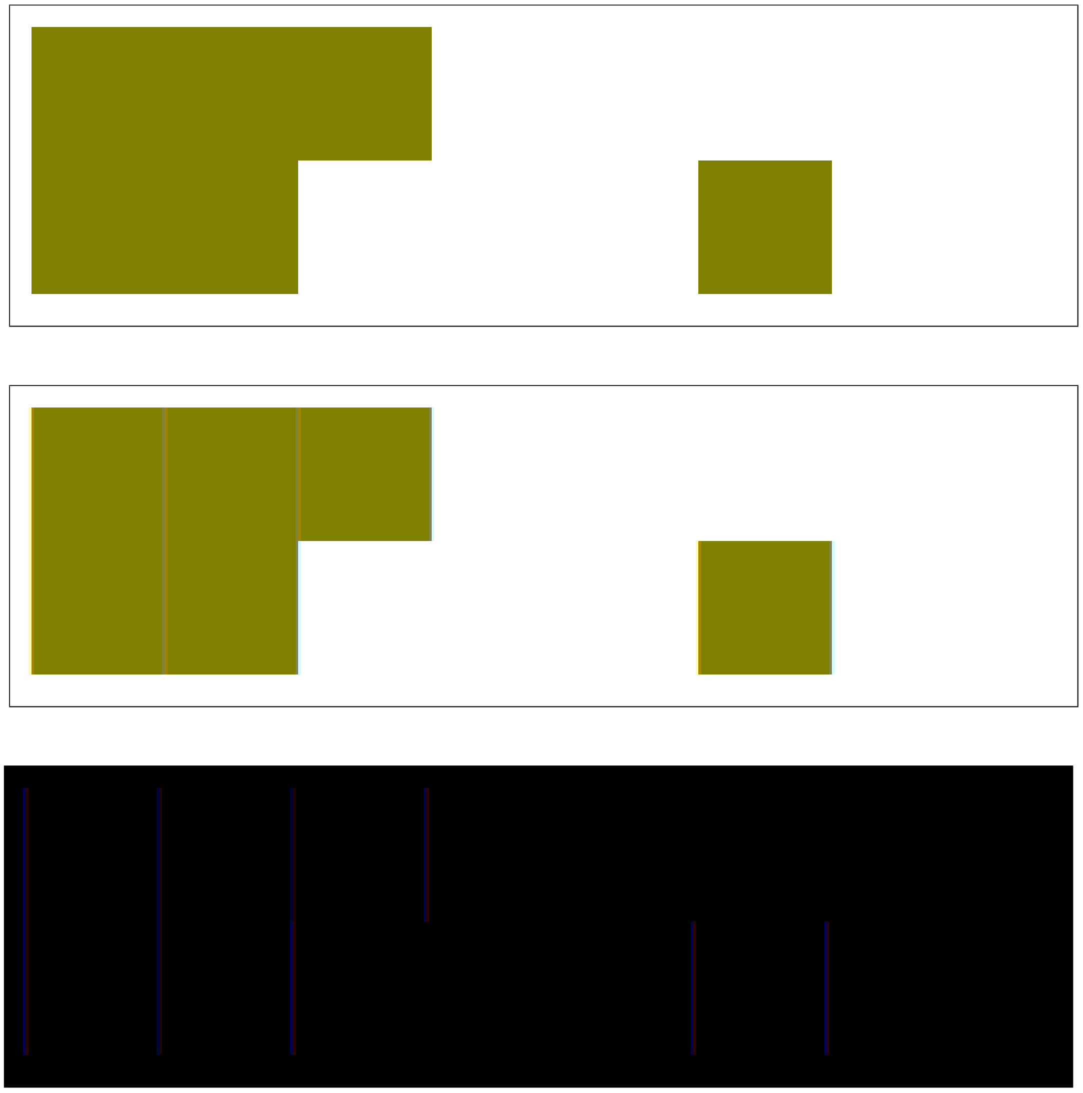


Figure 6.3: Rendering of HTML in Listing 6.8 in Firefox/Chrome, Servo, and as a color channel difference

browser. Different browser installations will likely have differing sets of available fonts. Even when the same size font is used, there may be differences in kerning and spacing of those fonts that can result in significant rendering differences. For example, two different font typefaces may result in page text occupying a different amount of space on the page which may cause other large elements to overflow and to render in a different spot on the web page.

The Ahem font [21] was designed specifically for the purpose of browser testing. The Ahem font consists of three different sized solid rectangles and empty space. Each is precisely specified so that they are able to be rendered the same regardless of platform and rendering engine [22]. The glyphs 'X' (Unicode 0x0058), 'p' (Unicode 0x0070), 'É' (Unicode 0x00c9) and ' ' (Unicode 0x0020) are reflective of the four possible Ahem font renderings. Figure 6.7 shows the string "Ahem test: Xp&#x00c9;" rendered in Chrome with and without the Ahem font.

During testing we discovered a bug with the rendering of the Ahem font in Servo that negatively impacts the ability to determine consensus. We reported this as Servo issue 24042 (`https://github.com/servo/servo/issues/24042`). The nature of this bug is that the left and right edge of each Ahem character is slightly transparent which allows the background color to be visible. Listing 6.8 shows a small test case that reproduces this bug. The top image in Figure 6.3 shows how this test case is rendered in Firefox and Chrome. The middle image shows the rendered result in Servo and it is clear that there are vertical lines between rendered Ahem characters. The bottom image is a color channel difference of the first and second images. Any differences between the two images is rendered as non-black pixels. The purple lines that are visible in the difference image shows that the vertical lines are not just occurring where rendered Ahem characters are directly adjacent but is also occurring on the left and right edges of those characters as well.

This issue with Ahem font rendering has not yet been resolved at the time the research was completed. In order to continue with testing we developed a mitigation for the issue. We modified the generator for character data to wrap every Ahem character with a `span`

```
/**
 * Add the correct display in IE 9-.
 */

article,
aside,
footer,
header,
nav,
section {
  display: block;
}

/**
 * Correct the font size and margin on `h1` elements within `section`
 * and `article` contexts in Chrome, Firefox, and Safari.
 */

h1 {
  font-size: 2em;
  margin: 0.67em 0;
}
```

Listing 6.10: A section of the normalize.css CSS Resets used in Bartender generated test cases

element that has a background color that matches the current foreground text color. Listing 6.9 shows the CSS style used for the tag and an example of the workaround.

We also discovered in early testing that in certain cases the test shrink phase was slow because it generated many small incremental tests cases while it was attempting to shrink text content. For this reason we limited the actual generated text characters list above. By limiting the text content grammar in this way we achieve a significant test shrink speedup for test cases that contain plain text content.

CSS Resets

There are a number of areas where the default rendering behaviors differ between browsers. These differences are not defects but rather an issue of differing configuration; each browser has a built-in default set of CSS styles (stylesheet) that is applied to all loaded pages. Bartender makes use of a technique called "CSS resets" [67] [68] to adjust CSS defaults to common values. Each test case generated by Bartender loads a stylesheet from the *normalize.css* project (version 7.0.0 [69]) to reset CSS styles. Listing 6.10 shows a portion of the *normalize.css* stylesheet. This part shows the display type for several HTML elements being adjusted (to compensate for a different default in version 9 of Microsoft's Internet Explorer browser) and the font size and margins being adjusted for the `<h1>` tag (to compensate for differences in Chrome, Firefox, and Apple Safari).

6.3.3 Consensus Example

This section describes the Bartender consensus process with example data from a actual test run. Listings 6.5 - 6.15 show the web page test cases (HTML and CSS) for several notable iterations (0, 2, 13, 14, 135, 171) from the test run. Figures 6.6 - 6.16 show screenshots and difference images that correspond to those listings. The figures are arranged to more clearly show which browsers the difference images related to by positioning each difference image between the pair of browser screenshots to which it relates.

Figure 6.4 shows a list view of the test run as represented by the monitoring application. A summary of the test run appears above the iteration list table. Each row of the table represents an iterations from a test run. The columns of the table are as follows: iteration number, overall iteration result (*PASS* or *FAIL*), test case link (HTML), screenshot image links (1 per browser), difference image links (1 per browser pair), average difference image link. The difference images are calculated by subtracting the color values of the screenshot image from one browser in the pair from the other in the pair. When there is no difference between two browsers the image will be solid black. A difference measure is then calculated for the difference images. The method of calculating the disagreement measurement value is configurable but defaults to calculating the SSD of each color channel and averaging them together to get a single number (described in more detail in Section 2.7.2). The difference image columns are highlighted with a red background if the disagreement measure exceeds the threshold for that pair. The browser column is highlighted with a red background if the browser is part of any browser pair where the difference value is above the threshold (the monitoring application is discussed in more detail in Section 6.5).

The test run starts with a small size test case in iteration 0. The HTML of the test case is shown in Listing 6.5 and contains only '`xX`' within the document body. There are no HTML attributes on the body tag and no additional HTML tags within the body. The rendered screenshots and difference images are shown in Figure 6.6. The test case renders identically in Firefox and Chrome (the disagreement measure is 0.0). The Servo rendering is very slightly different from the other two browsers however the disagreement measure of 0.000007 falls well below the configured threshold of 0.0001.

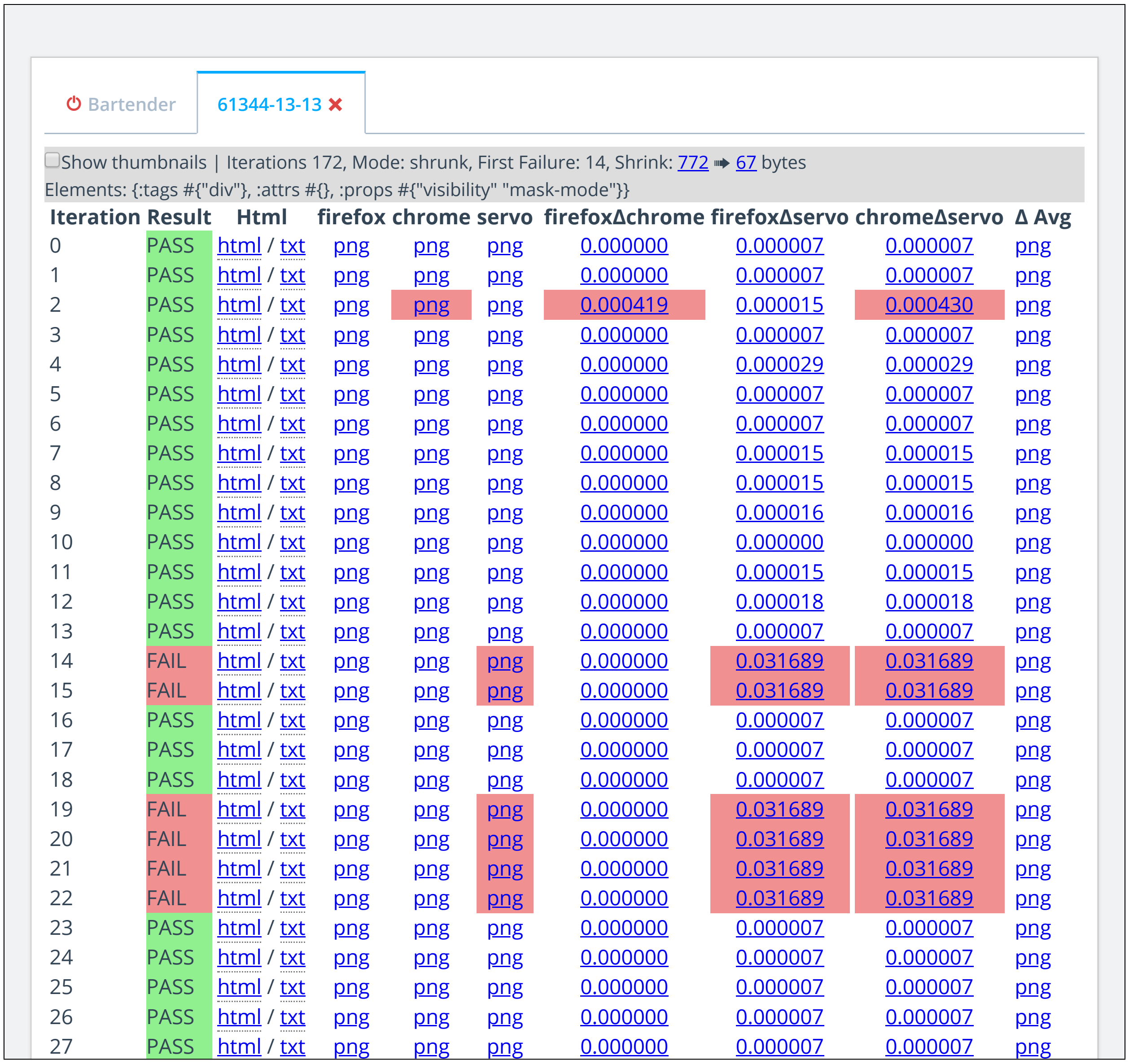
The first iteration where a disagreement measure exceeds the threshold is iteration 2. Listing 6.7 shows the test case HTML and Figure 6.8 shows the screenshots and difference images. This test case has a higher complexity (PBT test size) than the initial case. The body tag has CSS styles applied to it and there is a tag with an attribute in the body. Chrome has disagreement measures of 0.000419 with Firefox and of 0.000430 with Servo, both of which are above the threshold. Firefox and Servo have a disagreement measure of 0.000015 which is below the threshold. In other words, the Chrome rendering of the test case is reported as different from the other two browsers which are reported as the same. However, note that the overall iteration is not reported as a *FAIL*. The reason is that this test run was configured to performing targeted testing of Servo. This means that for any test case where Firefox and Chrome disagree on the rendering (difference measure above the threshold) the Servo data is ignored (*PASS*). In other words, in this configuration, Servo must be the only member of the pool that differs for the iteration to be considered a failure (counterexample). The effect of this configuration is that Firefox and Chrome are being used as the combined test oracle to test Servo.

Iteration 13 is the largest test case that is generated prior to a failure being reported. Listing 6.9 shows the test case HTML and Figure 6.10 shows the screenshots and difference images. In this test case the body contains both text and a `<div>` tag. The body and the `<div>` tag have 27 and 5 style properties attached respectively. No disagreement measures exceed the threshold so the test case is considered passing.

Iteration 14 represents an initial failure case where the rendering in Servo is different from Firefox and Chrome and where Firefox and Chrome are considered the same. Listing 6.11 shows the test case HTML and Figure 6.12 shows the screenshots and difference images. The test case is significantly large and more complex even than iteration 13. There are five different tags with the body. The body and two of the tags within the body have style properties applied to them. Text characters are scattered within and between the tags. It is clear from Subfigure 6.14e that Servo has a black rectangle in the upper left of screenshot while the other browsers do not. In fact, it appears from visual inspection that Firefox and Chrome are not showing any elements on the page apart from the background color on the document itself. However, it is not immediately obvious from the test case HTML what combination of tags, attributes, properties, or textual content is causing this difference in behavior.

Once the initial failing test case is found in iteration 14, the system transitions to the shrinking phase to try and find smaller and/or simpler versions of the test case that continue to fail. Listing 6.13 shows the test case HTML and Figure 6.14 shows the screenshots and difference images for iteration 135 during the shrinking process. This is an iteration that is identified as a failing case by the shrinking process (Servo is different from Firefox and Chrome). The test case is much simpler than the initial failure detected in iteration 14. The body has a single style property and contains a single `<param>` element with 13 CSS styles applied.

The shrinking process completes at iteration 171. Listing 6.15 shows the test case HTML and Figure 6.16 shows the screenshots and difference images for iteration 171. This is the minimal test case found by the system that was still reported as failing. This test case has an `x` character and an empty `<div>` tag within the body. The body and `<div>` tag each have a single CSS property applied to them. Compared to the original failure, the reduced test case greatly simplifies root cause analysis. In this case, searching for the CSS properties on Mozilla's Servo issue tracking system [70] reveals issue `https://github.com/servo/servo/issues/20142` showing that the `visibility: collapse` property is not yet implemented in Servo (as of 2019-11-15).



Bartender | 61344-13-13

Show thumbnails | Iterations 172, Mode: shrunk, First Failure: 14, Shrink: 772 ➡ 67 bytes
Elements: {:tags #{"div"}, :attrs #{}, :props #{"visibility" "mask-mode"}}

Iteration	Result	Html	firefox	chrome	servo	firefoxΔchrome	firefoxΔservo	chromeΔservo	Δ Avg
0	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
1	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
2	PASS	html / txt	png	png	png	0.000419	0.000015	0.000430	png
3	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
4	PASS	html / txt	png	png	png	0.000000	0.000029	0.000029	png
5	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
6	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
7	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
8	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
9	PASS	html / txt	png	png	png	0.000000	0.000016	0.000016	png
10	PASS	html / txt	png	png	png	0.000000	0.000000	0.000000	png
11	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
12	PASS	html / txt	png	png	png	0.000000	0.000018	0.000018	png
13	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
14	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
15	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
16	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
17	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
18	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
19	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
20	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
21	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
22	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
23	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
24	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
25	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
26	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
27	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png

Figure 6.4: Consensus test case example: list view

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="/static/normalize.css">
  <link rel="stylesheet" href="/static/rend.css">
  <title>
  </title>
</head>
<body>
  xX
</body>
</html>
```

Figure 6.5: Consensus test case example iteration 0: first iteration

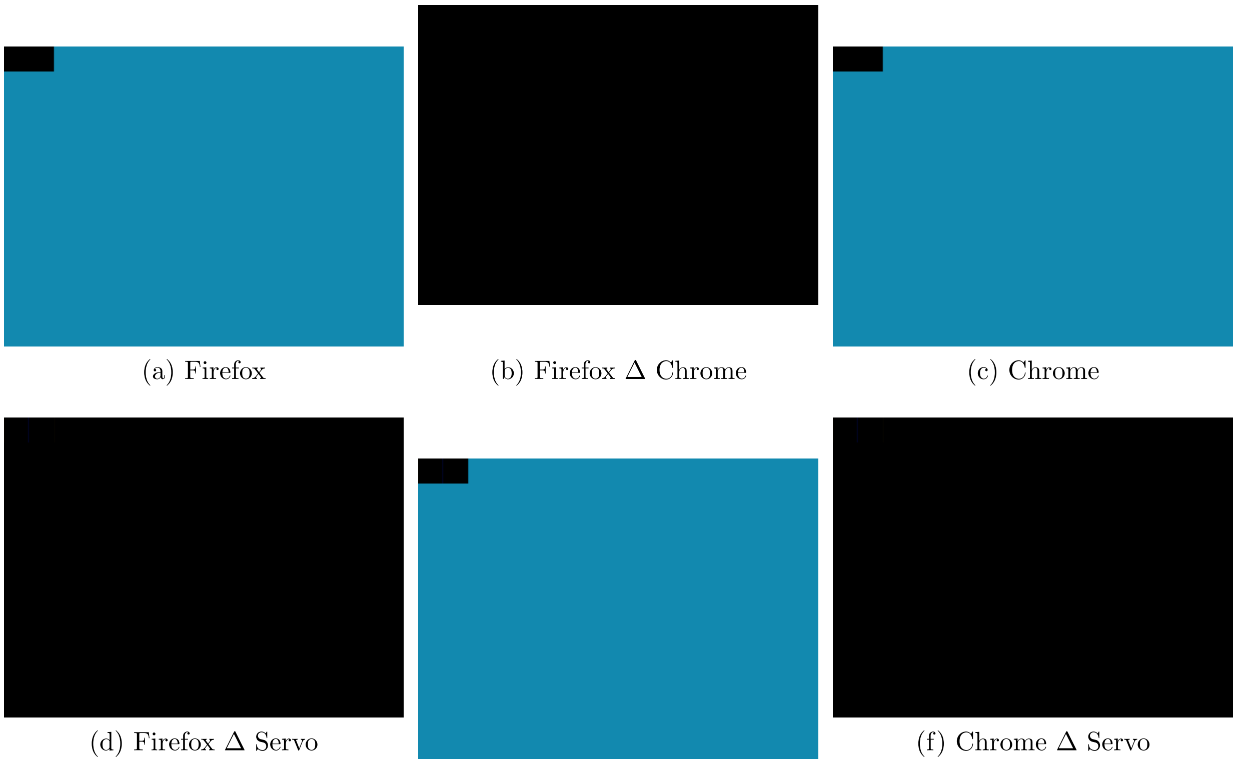

(a) Firefox (b) Firefox Δ Chrome (c) Chrome (d) Firefox Δ Servo (e) Servo (f) Chrome Δ Servo

Figure 6.6: Consensus test case example iteration 0 rendering: first iteration

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="/static/normalize.css">
  <link rel="stylesheet" href="/static/rend.css">
</head>
<body style="top : 1.0% ;
             line-height : 2 ;
             text-emphasis : revert ;
             transition-delay : unset">
  x
  <style type="u/x">
    X
  </style>
</body>
</html>
```

Figure 6.7: Consensus test case example iteration 2: Chrome difference ignored

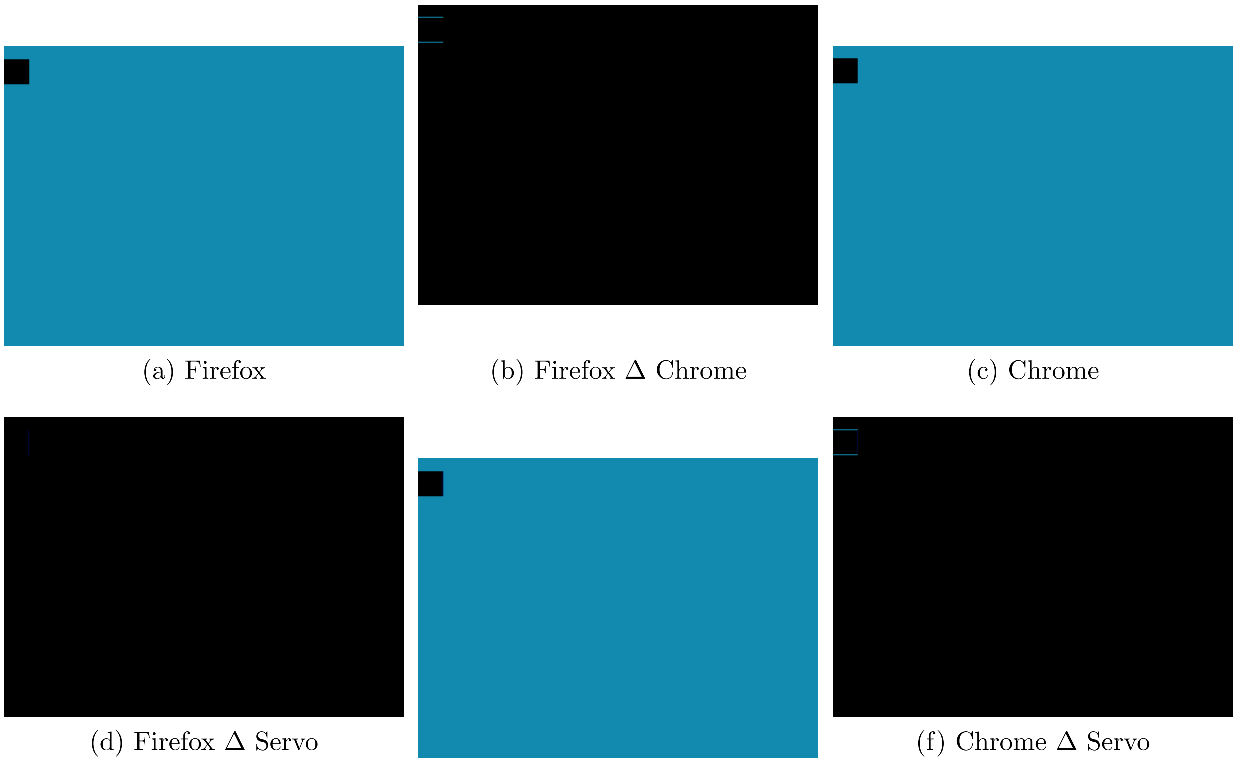
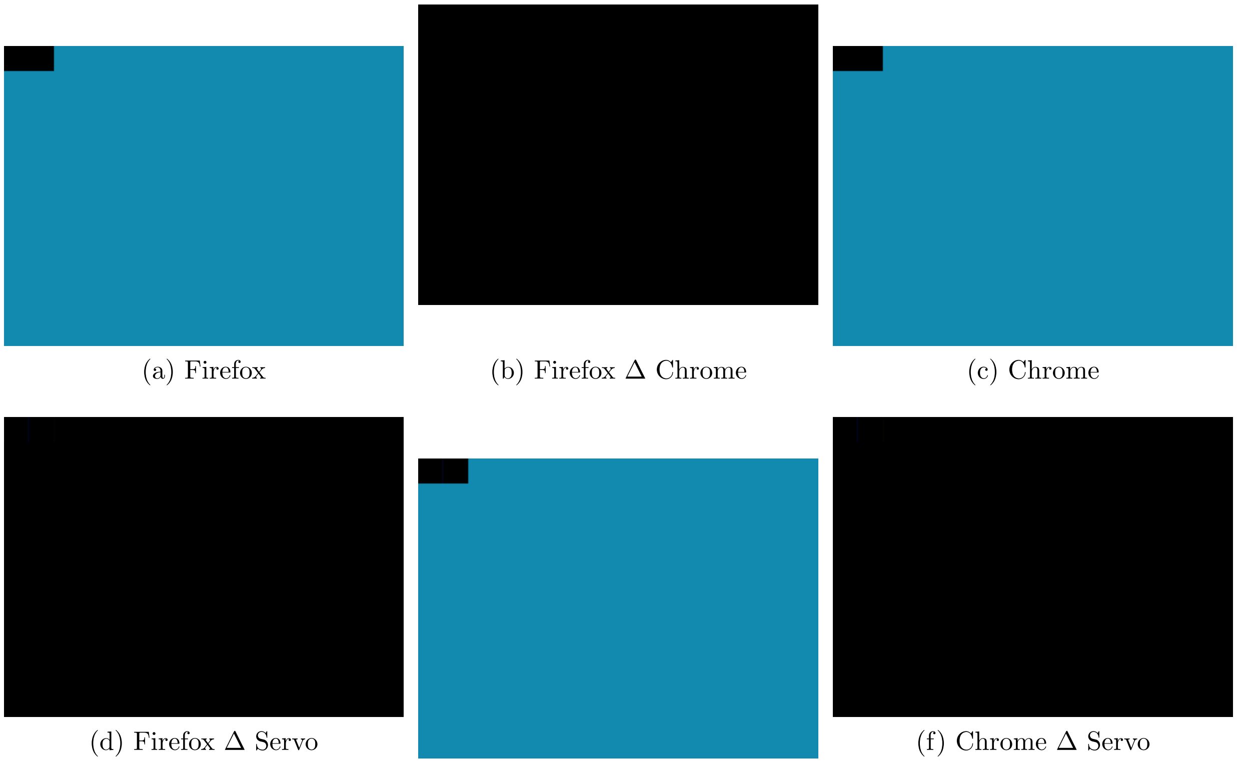

(a) Firefox
(b) Firefox Δ Chrome
(c) Chrome
(d) Firefox Δ Servo
(e) Servo
(f) Chrome Δ Servo

Figure 6.8: Consensus test case example iteration 2 rendering: Chrome difference ignored

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="/static/normalize.css">
  <link rel="stylesheet" href="/static/rend.css">
</head>
<body style="width : available ;
             outline-width : thick ;
             place-content : unset ;
             scroll-behavior : auto ;
             page-break-after : right ;
             justify-content : space-evenly ;
             white-space : unset ;
             scroll-margin-bottom : auto ;
             grid-auto-rows : revert ;
             outline-offset : -7em !important ;
             text-justify : none ;
             scroll-snap-align : unset ;
             shape-image-threshold : revert ;
             color-adjust : exact;
             place-items : unset ;
             place-self : unset ;
             text-justify : auto ;
             place-self : self-start auto ;
             transform-origin : left center 11vmin ;
             scroll-margin : unset ;
             z-index : 9;
             font-variant-position : unset ;
             text-justify : inter-word ;
             scroll-padding-inline-end : auto ;
             scroll-padding-bottom : -10em ;
             scroll-padding-block : unset ;
             unicode-bidi : isolate">
  xX
  <div style="list-style-image : url('QUOTED STRING') ;
              place-self : unset ;
              scroll-margin-block : unset ;
              scroll-padding-inline : auto 2ex;
              caption-side : inline-end ;
              /* CSS comment */">
  </div>
</body>
</html>
```

Figure 6.9: Consensus test case example iteration 13: prior to first failure


(a) Firefox

(b) Firefox Δ Chrome

(c) Chrome

(d) Firefox Δ Servo

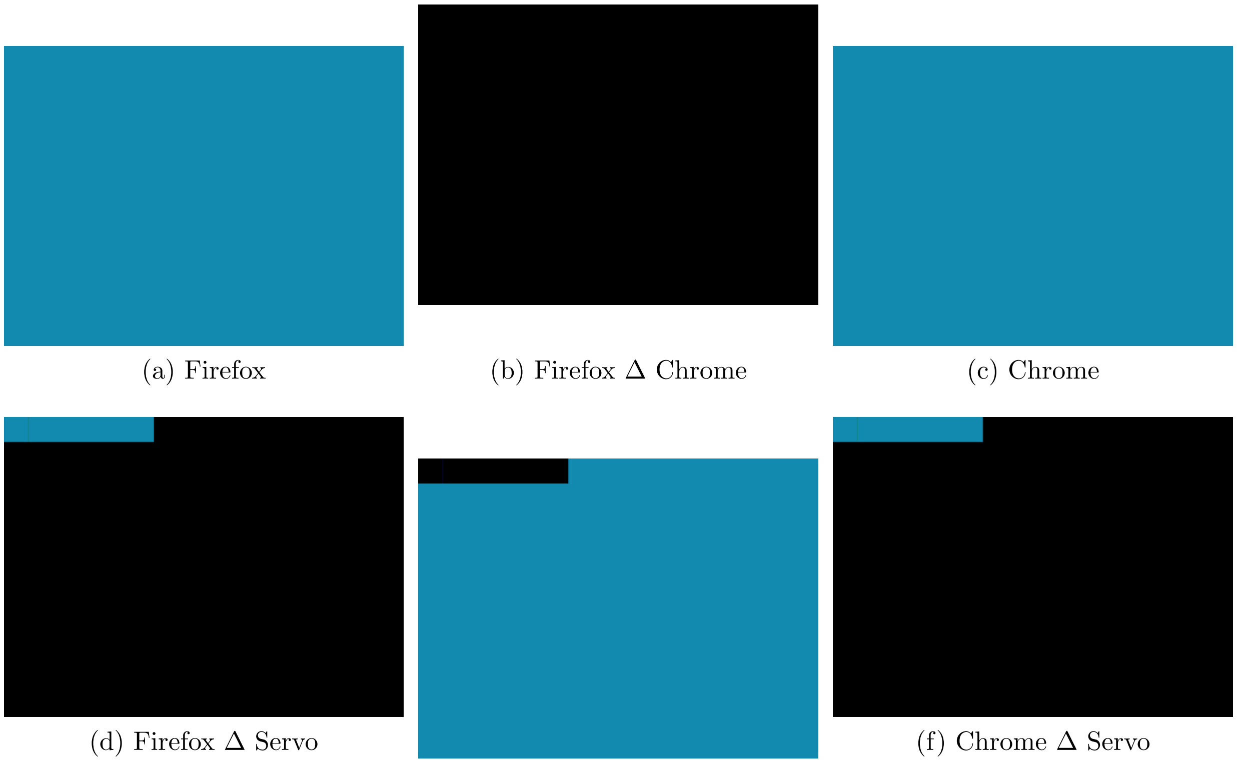
(e) Servo

(f) Chrome Δ Servo

Figure 6.10: Consensus test case example iteration 13 rendering: prior to first failure

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="/static/normalize.css">
  <link rel="stylesheet" href="/static/rend.css">
<title>
</title>
</head>
<body style="scroll-margin-inline-start : 10mm;
             page-break-after : left ;
             orphans : revert;
             visibility : collapse;
             border-image-source : revert;
             scale : -5.5"
      spellcheck="">
  x
  <cite dropzone=""
        style="vertical-align : top ;
               touch-action : manipulation ;
               scroll-margin-bottom : auto ;
               scroll-behavior : unset">
    X
    <!-- HTML comment -->
  </cite>
  <ins cite="">
  </ins>
  <script>
    XXXX
    <!-- HTML comment -->
  </script>
  XXX
  <tfoot style="">
  </tfoot>
  <template>
    XX
  </template>
  X
  <rp lang=""
      style="min-block-size : 6rem;
             outline-color : rgb ( -6% -6% -3.9154052734375%) ;
             scroll-margin-left : 5mozmm ;
             scroll-behavior : smooth ;
             shape-image-threshold : 3;
             shape-image-threshold : revert ;
             border-style : unset;
             animation-iteration-count : infinite;
             text-indent : 2%"
      tabindex="5"
      title="">
  </rp>
</body>
</html>
```

Figure 6.11: Consensus test case example iteration 14: initial failure



(a) Firefox

(b) Firefox Δ Chrome

(c) Chrome

(d) Firefox Δ Servo

(e) Servo

(f) Chrome Δ Servo

Figure 6.12: Consensus test case example iteration 14 rendering: initial failure

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="/static/normalize.css">
  <link rel="stylesheet" href="/static/rend.css">
</head>
<body style="visibility : collapse">
  x
  <param value="q" name="" name="wvzxnofDH" name="" name="zi"
         style="unicode-bidi : plaintext;
                scroll-padding-top : 1.75% ;
                opacity : unset ;
                pointer-events : inherit ;
                scroll-margin-top : auto ;
                transition-delay : revert ;
                object-fit : revert ;
                opacity : unset ;
                place-content : baseline normal ;
                place-content : revert ;
                padding-inline-start : revert ;
                orphans : unset;
                resize : block">
</body>
</html>
```

Figure 6.13: Consensus test case example iteration 135: shrinking

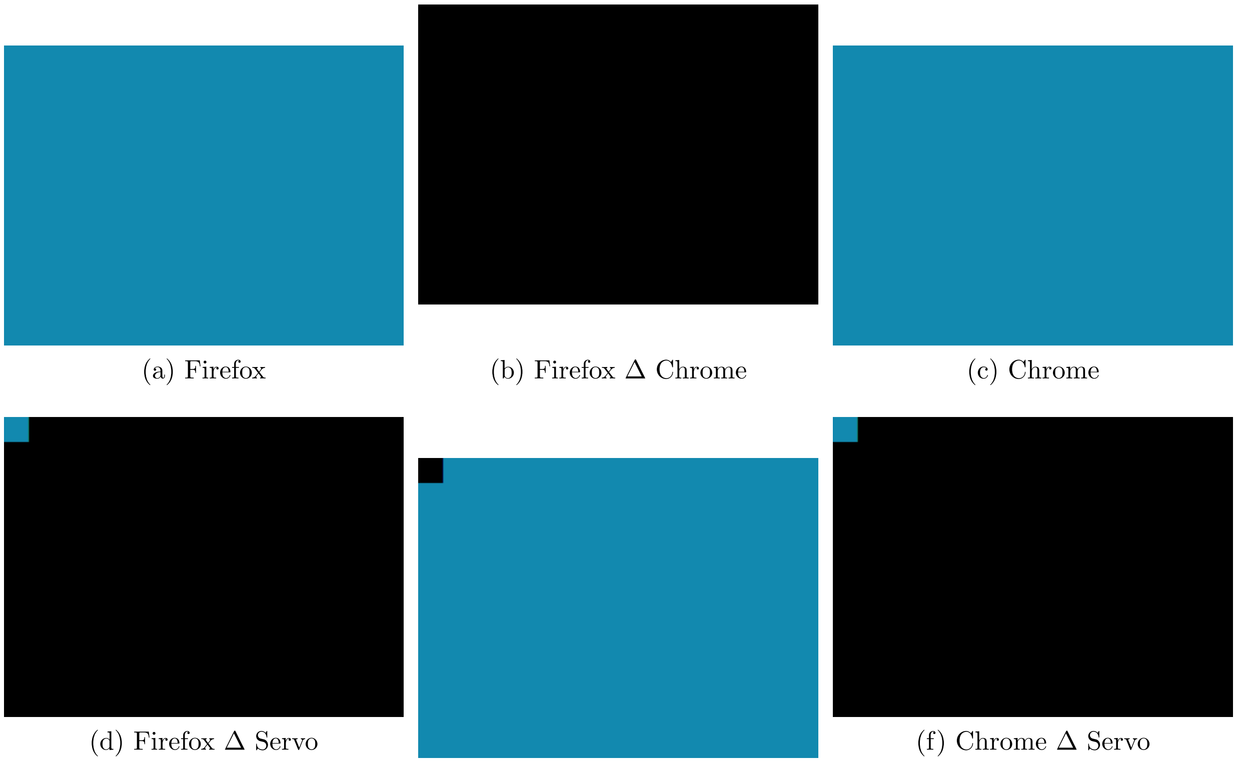


Figure 6.14: Consensus test case example iteration 135 rendering: shrinking

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="/static/normalize.css">
  <link rel="stylesheet" href="/static/rend.css">
</head>
<body style="visibility : collapse">
  x
  <div style="mask-mode : unset">
  </div>
</body>
</html>
```

Figure 6.15: Consensus test case example iteration 171: shrunk

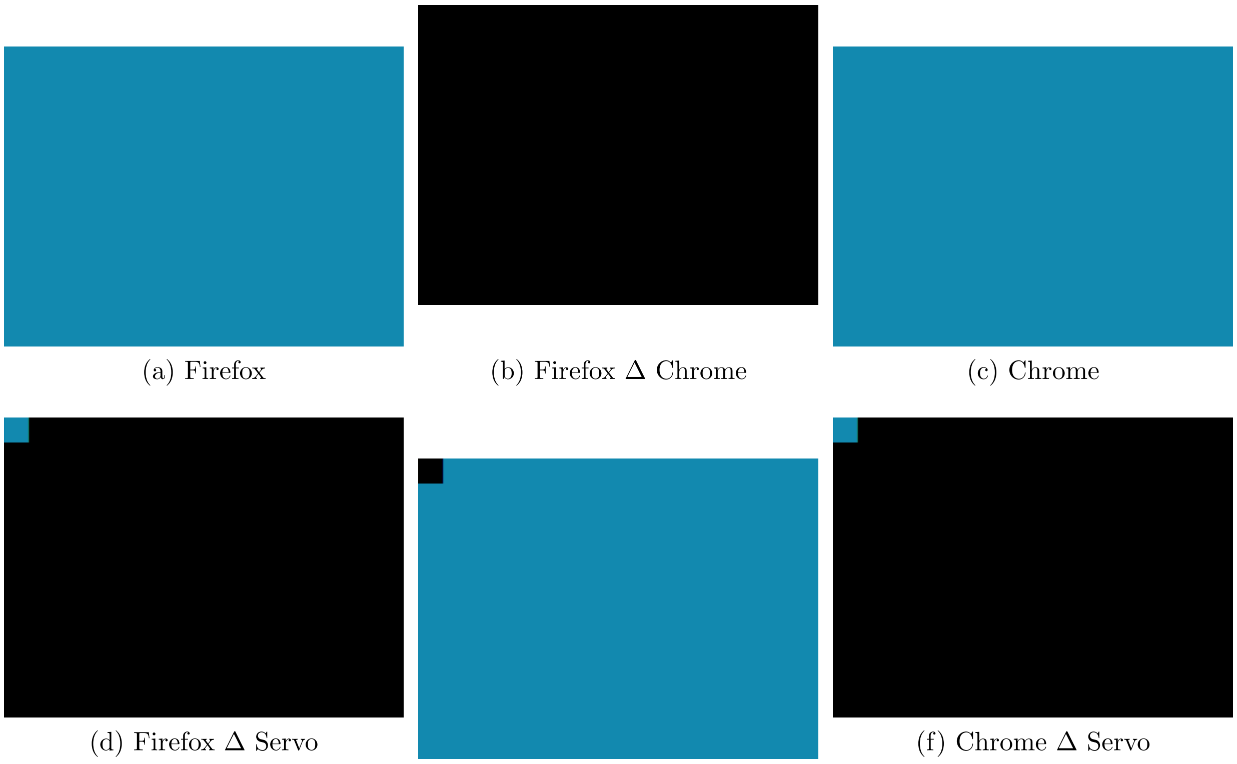


Figure 6.16: Consensus test case example iteration 171 rendering: shrunk

6.4 WebDriver Browser Automation

Bartender uses the WebDriver [13] framework to automate the browsers that are part of the consensus test pool. The WebDriver specification is being standardized by the W3C and derived from the Selenium framework [71]. Each browser in the consensus pool starts in a special testing mode that provides HyperText Transfer Protocol (HTTP) endpoints for the WebDriver interface. The WebDriver specification defines dozens of endpoints but Bartender only needs to interact with six endpoints in order to accomplish the automation required for browser render testing Table 6.1 lists the WebDriver HTTP endpoints that are used by Bartender and what they are used for.

Table 6.1: WebDriver endpoints

HTTP Endpoint	HTTP Method	Description
/session	POST	Create a new session and return SESSION-ID
/session/{SESSION-ID}	DELETE	Delete an existing session
/session/{SESSION-ID}/url	POST	Load a new web page by URL
/session/{SESSION-ID}/execute/sync	POST	Execute JS and return the result
/session/{SESSION-ID}/window/rect	POST	Alter the browser window size
/session/{SESSION-ID}/screenshot	GET	Take a screenshot and

When Bartender starts up it creates a connection to each browser in the consensus pool by making a HTTP *POST* request to the */session* WebDriver endpoint (6.1-I). This request includes "capabilities" information which is basically constraints that the client (Bartender) is requesting from the browser. If the browser can satisfy these constraints then the WebDriver interface creates a context for a new automation session and returns a unique ID *SESSION-ID* so that the caller can access the state of the session in subsequent calls. When Bartender terminates, either at the end of a series of test runs or due to some exception condition, an HTTP *DELETE* request is sent to the */session/*{*SESSION-ID*} endpoint to terminate the session and allow the browser to free up any windows or resources associated with the session (where {*SESSION-ID*} is a variable representing the session ID of the session).

For determining consensus it is important that the browsers have a common size for their rendering viewport. This is not just so that the screenshot images are the same size, but also because the size of the available rendering area can change the way that the same web page test case is rendered. For this reason, once Bartender has created a WebDriver session, it then sends an HTTP *POST* request to the */session/{SESSION-ID}/execute/sync* endpoint containing a small piece of JavaScript (JS) code that is executed by the browser to determine the size of the browsers control elements (menus, control bar, scroll bar, status bars, etc). Once this information is returned to Bartender it can use this information to calculate the desired browser window size so that the rendering viewport is the same on all browsers. To adjust the browser window size an HTTP *POST* request is send to the */session/{SESSION-ID}/window/rect* endpoint with the desired window dimensions.

During the testing process, Bartender sends an HTTP *POST* request to the */session/{SESSION-ID}/url* endpoint with the Uniform Resource Locator (URL) of the current test case. Then an HTTP *GET* request is sent to the */session/{SESSION-ID}/screenshot* endpoint to capture a screenshot of the current rendered state.

In addition to browsers started locally to be part of a consensus pool, Bartender also support browsers running on remote WebDriver-based services such as BrowserStack [72]. Listing G.3 shows a portion of the YAML (Yet Another Markup Language) configuration for Bartender that is used to configure three different browsers in the BrowserStack service. Note that the *capabilities* field is used as part of the BrowserStack session creation HTTP *POST* request to specify the requested browser vendor, browser version, browser window geometry, etc.

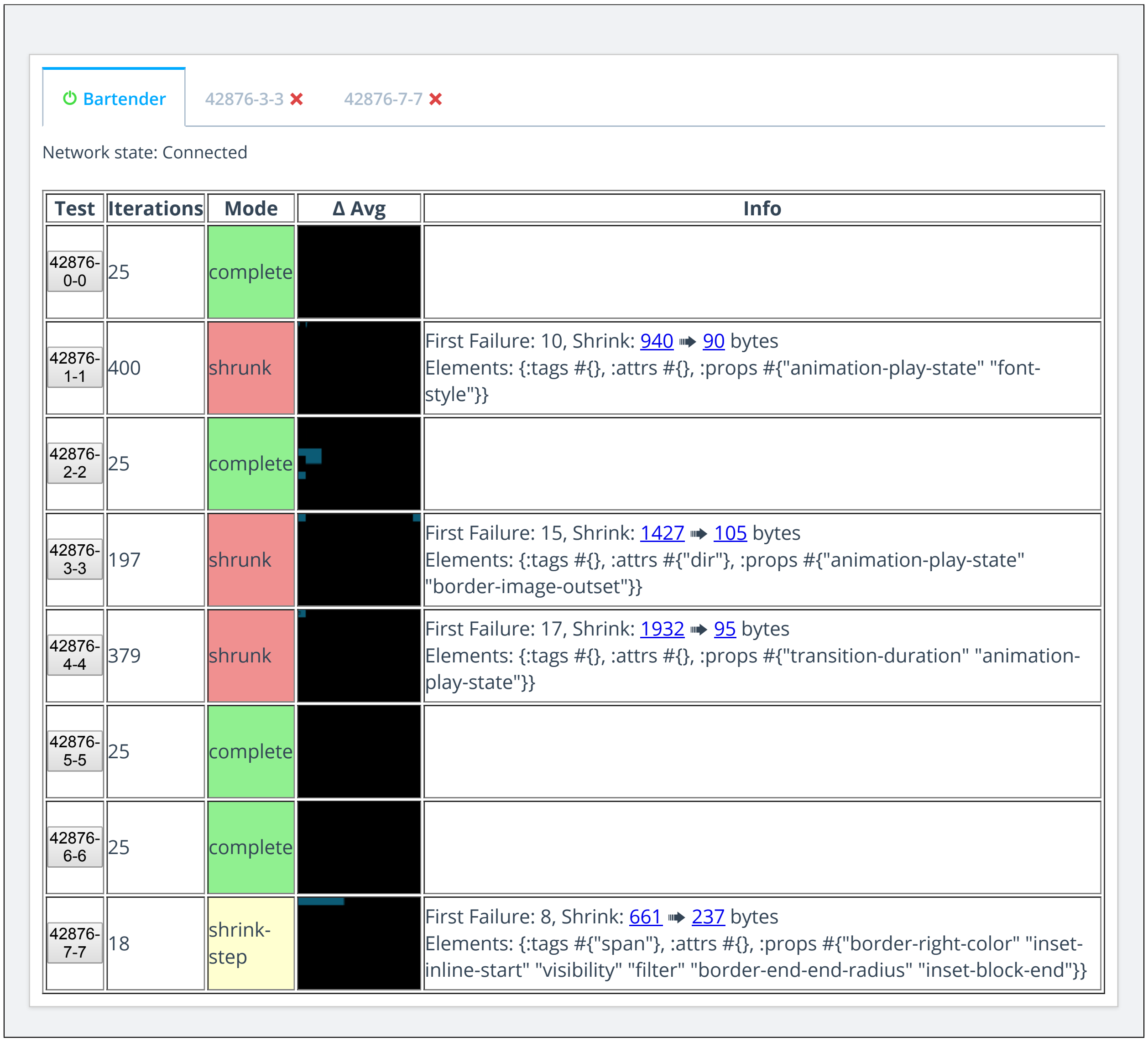
6.5 Monitoring and Reporting Applications

One of the central goals of the Bartender system is to make the underlying testing approach accessible and practical for users of the system. Towards this end, Bartender includes built-in web applications that provide runtime monitoring of test runs and provide summary reports of the test results. These applications are implemented with ClojureScript and use the Reagent ClojureScript interface [73] to the React [74] library for generating the user interface and dynamically rendering the test state data.

6.5.1 Runtime Monitoring Application

Bartender provides runtime monitoring via a web application that is updated dynamically as test runs are executed. The initial application resources (HTML, CSS, and images) are loaded from the embedded web server. Once the application is loaded, it initializes a WebSocket connection back to the Bartender system via the embedded webserver. Bartender registers this new WebSocket client and sends an initial message containing a complete copy of Bartender's runtime state including the test configuration, current active wtrek, and the full log of every run and iteration. The Bartender runtime state can become fairly large over multiple test runs so after the initial full state message, Bartender sends delta messages rather than full copies of the state. These delta messages contain change set structures that are generated using the *differ* [75] Clojure library. On the web application side, these change set (diff) structures are then used to patch the runtime state. The combination of WebSockets and delta messages means that that web application state can efficiently keep in sync with Bartender server state.

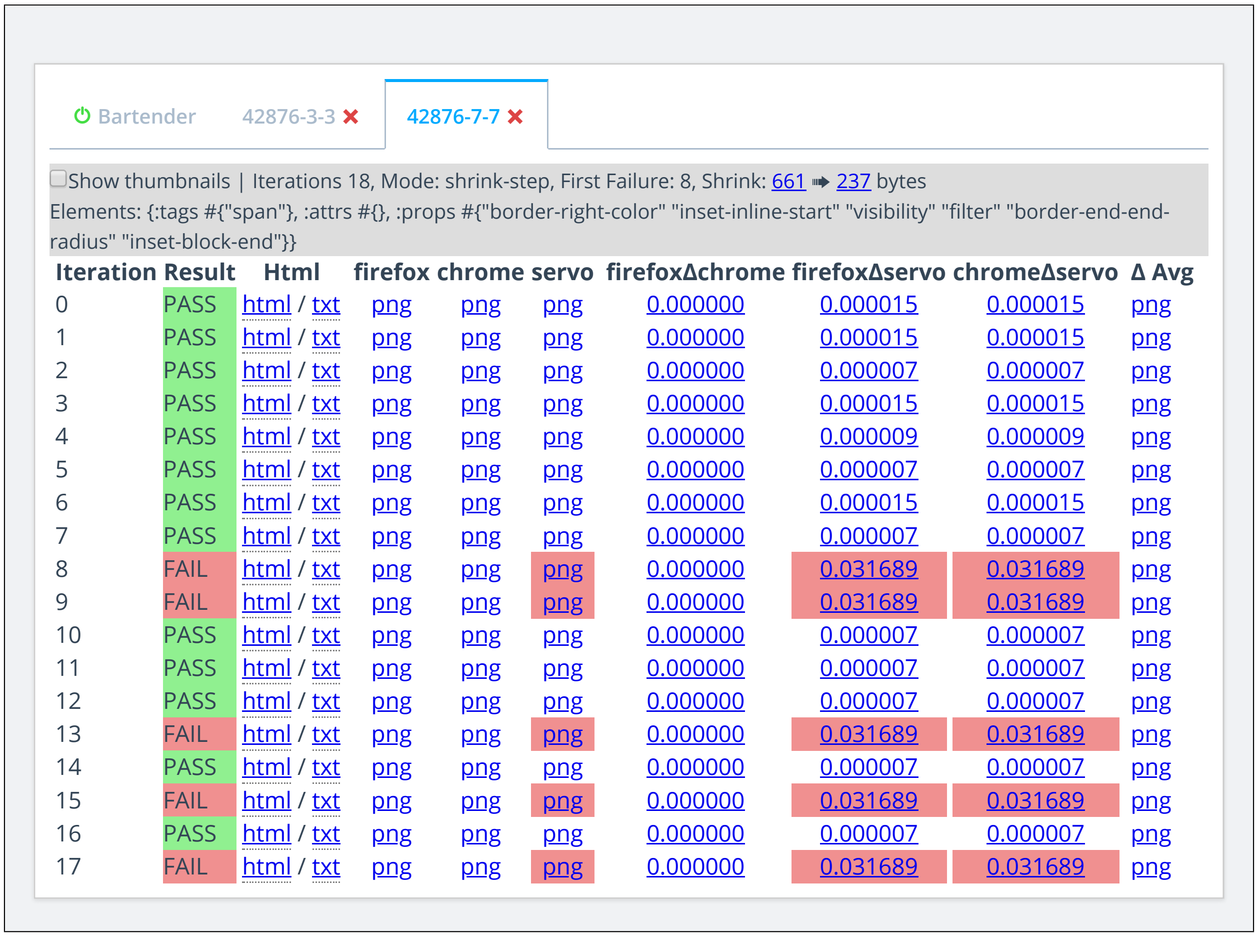
Figure 6.17 shows the initial view from the reporting application. At the top of the application is a tabbed navigation bar that is used to select the information the user wishes to view. The left-most tab indicates that the application has an active WebSocket connection and is able to receive dynamic updates from the Bartender system ("Network state: Con-



Network state: Connected

Test	Iterations	Mode	Δ Avg	Info
42876-0-0	25	complete		
42876-1-1	400	shrunk		First Failure: 10, Shrink: 940 ➠ 90 bytes Elements: {:tags #{}, :attrs #{}, :props #{"animation-play-state" "font-style"}}
42876-2-2	25	complete		
42876-3-3	197	shrunk		First Failure: 15, Shrink: 1427 ➠ 105 bytes Elements: {:tags #{}, :attrs #{"dir"}, :props #{"animation-play-state" "border-image-outset"}}
42876-4-4	379	shrunk		First Failure: 17, Shrink: 1932 ➠ 95 bytes Elements: {:tags #{}, :attrs #{}, :props #{"transition-duration" "animation-play-state"}}
42876-5-5	25	complete		
42876-6-6	25	complete		
42876-7-7	18	shrink-step		First Failure: 8, Shrink: 661 ➠ 237 bytes Elements: {:tags #{"span"}, :attrs #{}, :props #{"border-right-color" "inset-inline-start" "visibility" "filter" "border-end-end-radius" "inset-block-end"}}

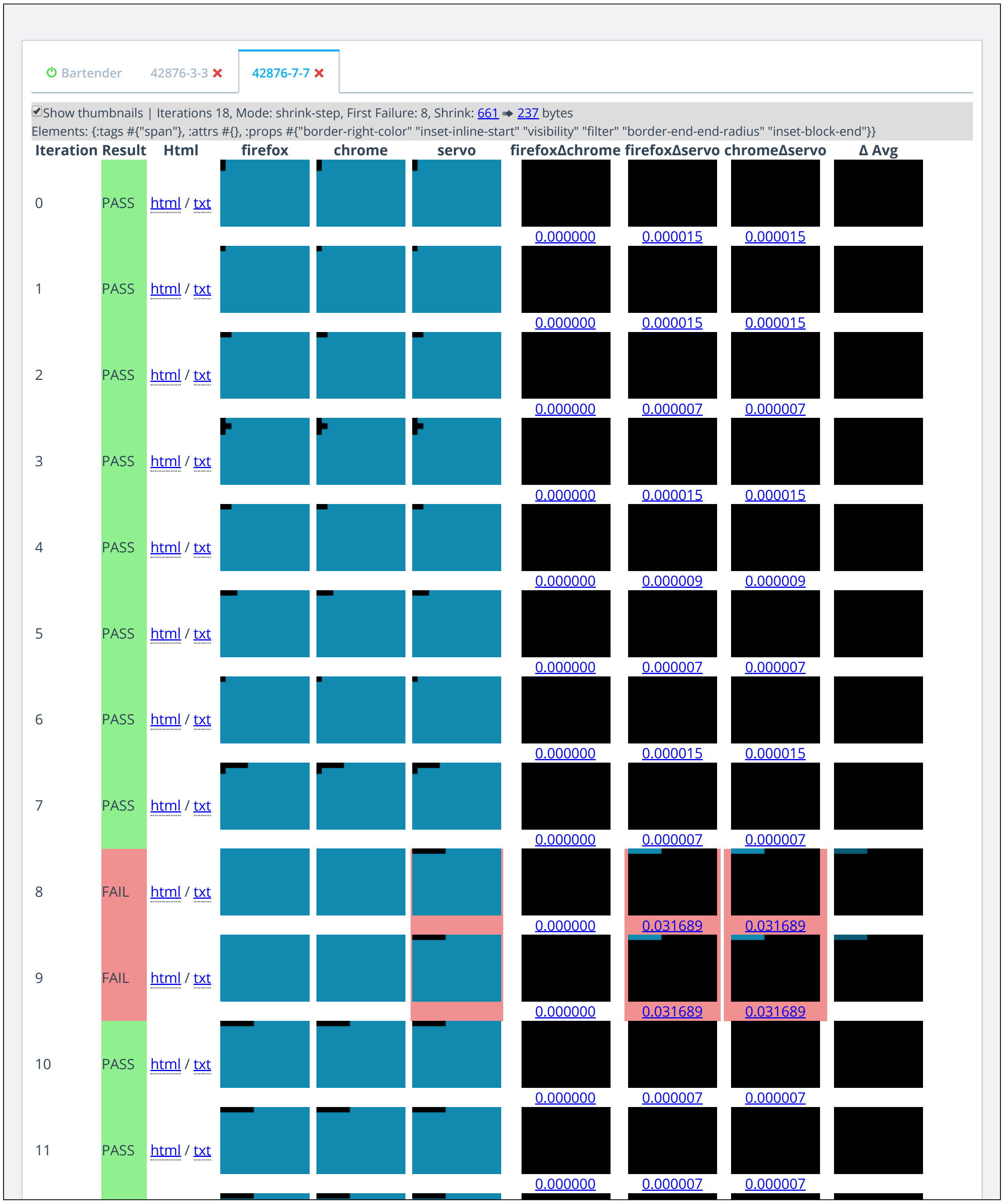
Figure 6.17: Monitoring application main tab view



Show thumbnails | Iterations 18, Mode: shrink-step, First Failure: 8, Shrink: 661 ➠ 237 bytes
Elements: {:tags #{"span"}, :attrs #{}, :props #{"border-right-color" "inset-inline-start" "visibility" "filter" "border-end-end-radius" "inset-block-end"}}

Iteration	Result	Html	firefox	chrome	servo	firefoxΔchrome	firefoxΔservo	chromeΔservo	Δ Avg
0	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
1	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
2	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
3	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
4	PASS	html / txt	png	png	png	0.000000	0.000009	0.000009	png
5	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
6	PASS	html / txt	png	png	png	0.000000	0.000015	0.000015	png
7	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
8	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
9	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
10	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
11	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
12	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
13	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
14	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
15	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png
16	PASS	html / txt	png	png	png	0.000000	0.000007	0.000007	png
17	FAIL	html / txt	png	png	png	0.000000	0.031689	0.031689	png

Figure 6.18: Monitoring application test run view


Bartender 42876-3-3 42876-7-7

Show thumbnails | Iterations 18, Mode: shrink-step, First Failure: 8, Shrink: 661 ➡ 237 bytes
Elements: {:tags #{"span"}, :attrs #{}, :props #{"border-right-color" "inset-inline-start" "visibility" "filter" "border-end-end-radius" "inset-block-end"}}

Iteration	Result	Html	firefox	chrome	servo	firefoxΔchrome	firefoxΔservo	chromeΔservo	Δ Avg
0	PASS	html / txt				0.000000	0.000015	0.000015	
1	PASS	html / txt				0.000000	0.000015	0.000015	
2	PASS	html / txt				0.000000	0.000007	0.000007	
3	PASS	html / txt				0.000000	0.000015	0.000015	
4	PASS	html / txt				0.000000	0.000009	0.000009	
5	PASS	html / txt				0.000000	0.000007	0.000007	
6	PASS	html / txt				0.000000	0.000015	0.000015	
7	PASS	html / txt				0.000000	0.000007	0.000007	
8	FAIL	html / txt				0.000000	0.031689	0.031689	
9	FAIL	html / txt				0.000000	0.031689	0.031689	
10	PASS	html / txt				0.000000	0.000007	0.000007	
11	PASS	html / txt				0.000000	0.000007	0.000007	

Figure 6.19: Monitoring application test run view with thumbnail images

nected"). Below the connection state indicator is a table containing a summary of each test run that has either been completed or is currently executing.

Figure 6.17 shows seven completed test runs and one test run that is currently executing. The table columns are:

1. The **test** run slug consisting of a random ID for this set of test runs, the test run index, and the starting random seed for the test set.
2. The number of **iterations** completed for the test run.
3. The current **mode**, or state of the test run ("trial", "complete", "failure", "shrink-step", "shrunk")
4. A thumbnail image of the average of all difference images (**Δ Avg**).
5. Summary **info** of the test run including the initial and smallest shrunk byte size and a summary of the HTML tags, HTML attributes, and CSS property names that appear in the smallest test case discovered so far.

In Figure 6.17 four of the test runs completed 25 iterations without finding an initial failure (indicated by the green "complete" mode). Three test runs found an initial failure and completed the shrinking process (indicated by the red "shrunk" mode). The currently executing test run found an initial failure and is currently performing the shrinking process (indicated by the "shrink-step" mode). From this view the tester can click on a test run slug button to add that test run to the tabbed navigation bar. When the user clicks on the test run slug tab, the test run view is show for that test run. Figure 6.18 shows the test run view for the currently executing test run. The test run table columns are:

1. The **iteration** index.
2. The **result** of the iteration ("PASS" or "FAIL")
3. Links to the **HTML** test case (HTML form and raw text form)

4. A label for each browser in the consensus pool: **firefox**, **chrome**, and **servo**. Browsers with any difference measures exceeding the threshold value are highlighted with a red background.

5. The difference measure of the screenshots and links to the screenshots for each browser pair: **firefoxΔchrome**, **firefoxΔchrome**, and **firefoxΔservo**. Difference measures exceeding the threshold value are highlighted with a red background.

6. A link to an average of the screenshot images (Δ **Avg**).

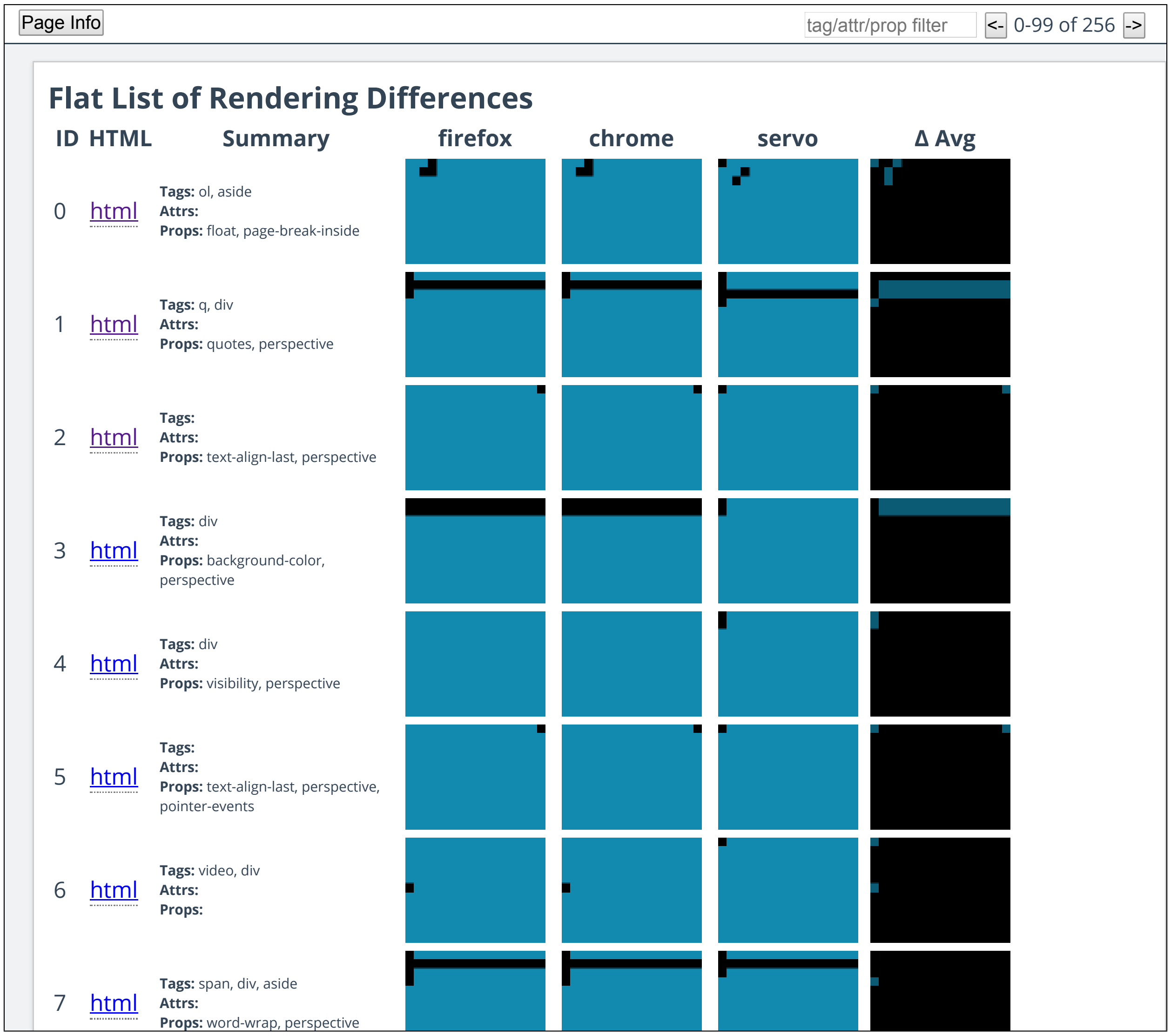
The test run shown in Figure 6.18 detected a failure on the ninth iteration (iteration index 8) and had completed 9 shrinking iterations when the screenshot in the figure was captured. The top of the test run view has the summary information for this test run. Figure 6.19 shows the same test run view with the thumbnail images enabled.

6.5.2 Test Reporting Application

In addition to the runtime monitoring application described in Section 6.5.1, Bartender also provides a web application for loading and reporting test run results. The reporting application loads one or more Bartender test run log files and provides two different views of the test results. The first view provides a flat list of test cases that can be filtered by HTML tag, HTML attribute, or CSS property name. Figure 6.20 shows an example of the view with a test run log file loaded that contains the data for 256 failing test cases. Figure 6.21 shows the same data filtered to only test cases that the included the CSS *float* property name. The columns of the results table are:

1. A unique **ID** index assigned to each failing test case. This can be used to correlate results with results from the table view reporting application.

2. A Link to the **HTML** test case.

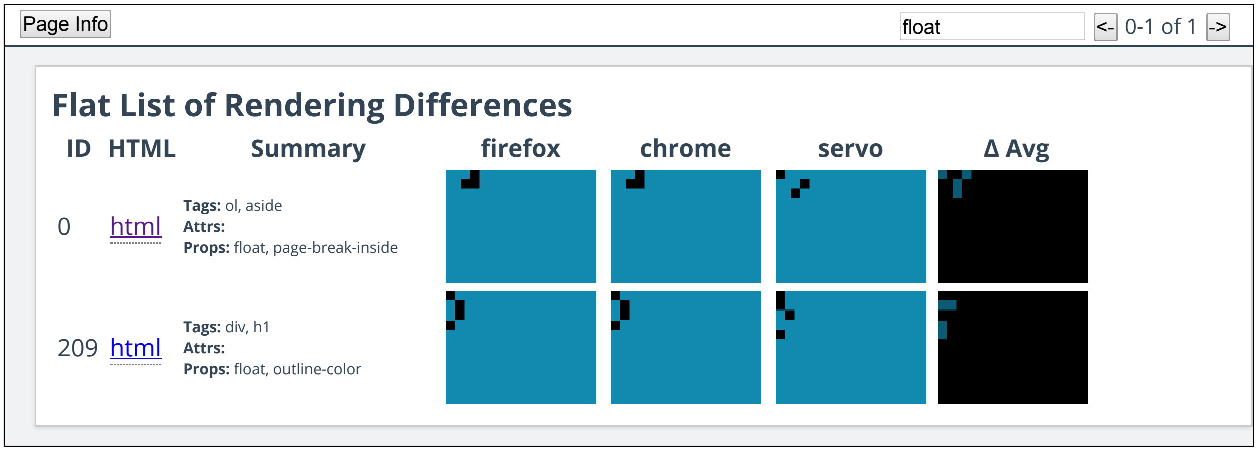


Page Info   tag/attr/prop filter   <- 0-99 of 256 ->

Flat List of Rendering Differences

ID	HTML	Summary	firefox	chrome	servo	Δ Avg
0	html	**Tags:** ol, aside **Attrs:** **Props:** float, page-break-inside				
1	html	**Tags:** q, div **Attrs:** **Props:** quotes, perspective				
2	html	**Tags:** **Attrs:** **Props:** text-align-last, perspective				
3	html	**Tags:** div **Attrs:** **Props:** background-color, perspective				
4	html	**Tags:** div **Attrs:** **Props:** visibility, perspective				
5	html	**Tags:** **Attrs:** **Props:** text-align-last, perspective, pointer-events				
6	html	**Tags:** video, div **Attrs:** **Props:**				
7	html	**Tags:** span, div, aside **Attrs:** **Props:** word-wrap, perspective				

Figure 6.20: Reporting application flat list view showing first page of test results


Page Info

float <- 0-1 of 1 ->

Flat List of Rendering Differences

ID	HTML	Summary	firefox	chrome	servo	Δ Avg
0	html	**Tags:** ol, aside **Attrs:** **Props:** float, page-break-inside				
209	html	**Tags:** div, h1 **Attrs:** **Props:** float, outline-color				

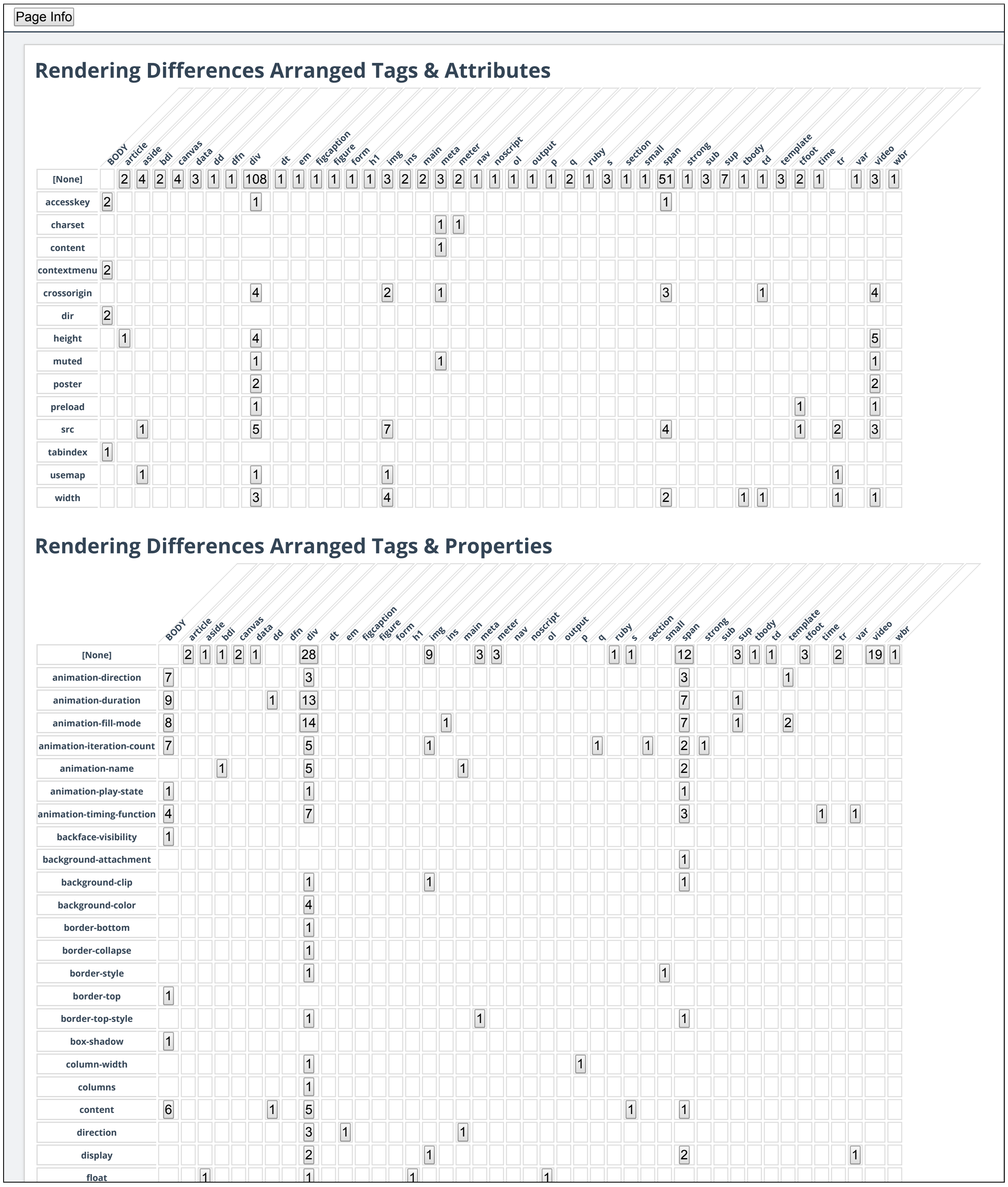
Figure 6.21: Reporting application flat list view showing only tests that contain the float property name

3. A **Summary** of the HTML tags, HTML attributes, and CSS property names that appear in the test case.

4. A screenshot thumbnail image for each browser that was in the consensus pool when test case was discovered: **firefox**, **chrome**, and **servo**.

5. A thumbnail image of the average of the screenshot images (Δ **Avg**).

The second reporting application presents two tables with a matrix of test results arrange by HTML Tags and HTML attributes and by HTML tags and CSS property names. Figure 6.22 shows an example table/matrix view of the same 256 test cases that were described in the flat list view described above. An non-empty cell in each table represents the intersection of row and column features where one or more failing test cases were discovered. The count on the button represents the number of test cases that matched those features, while an empty cell means the loaded data contains no failing test cases that match those features. If the user clicks on a cell button then the application shows a modal dialog containing the test cases that matched those features. An example of this is shown in Figure 6.23 for the *img* tag and *src* attribute. Note that the modal dialog provides the same information and layout as the flat list view but filtered by the feature of the selected cell from the matrix


Page Info

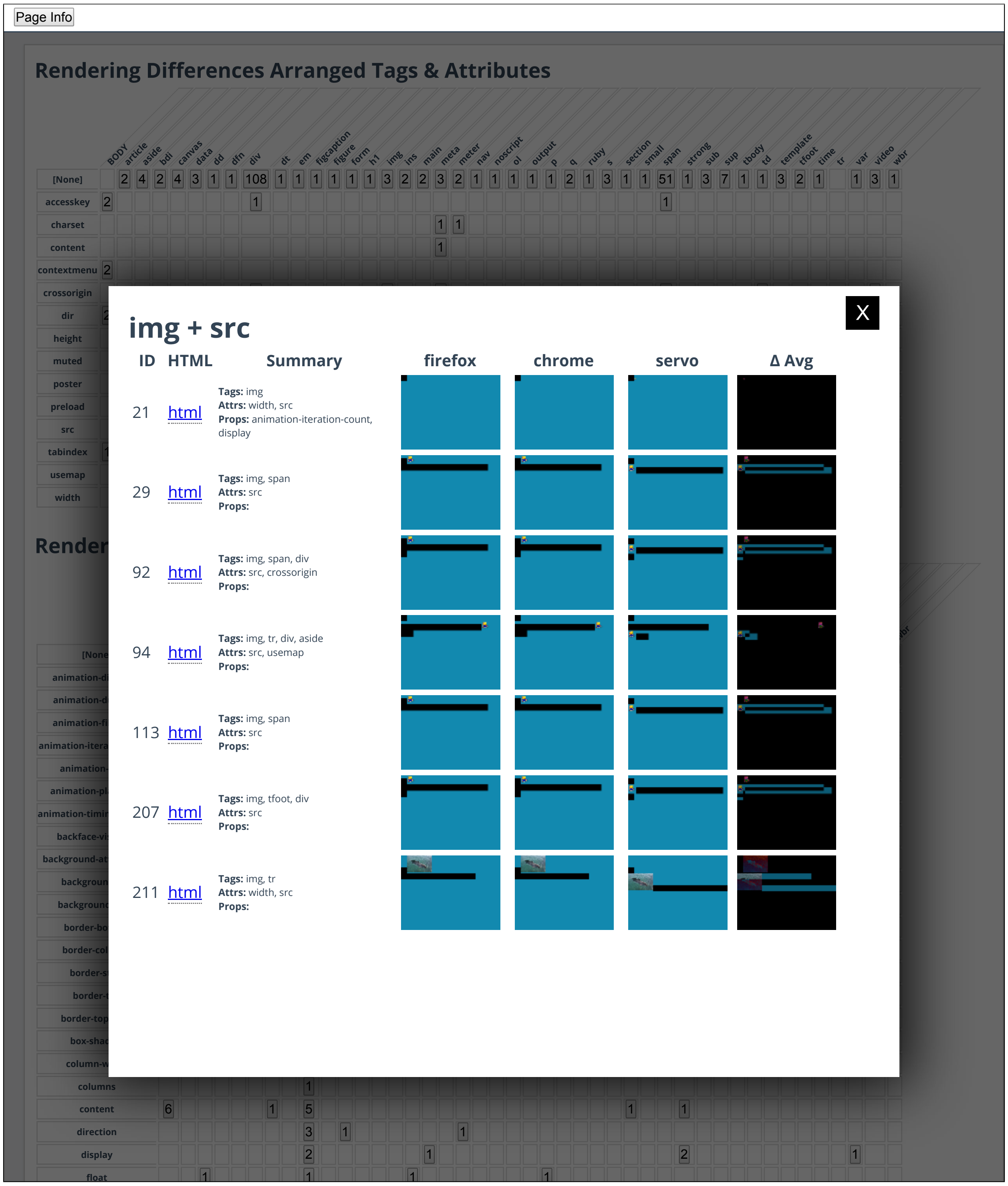
Rendering Differences Arranged Tags & Attributes

	BODY	article	aside	bdi	canvas	data	dd	dfn	div	dt	em	figcaption	figure	form	h1	img	ins	main	meta	meter	nav	noscript	ol	output	p	q	ruby	s	section	small	span	strong	sub	sup	tbody	td	template	tfoot	time	tr	var	video	wbr
[None]		2	4	2	4	3	1	1	108	1	1	1	1	1	1	3	2	2	3	2	1	1	1	1	1	2	1	3	1	1	51	1	3	7	1	1	3	2	1		1	3	1
accesskey	2								1																						1												
charset																			1	1																							
content																			1																								
contextmenu	2																																										
crossorigin									4							2			1												3					1						4	
dir	2																																										
height		1							4																																	5	
muted									1										1																							1	
poster									2																																	2	
preload									1																													1				1	
src			1						5							7															4							1		2		3	
tabindex	1																																										
usemap			1						1							1																								1			
width									3							4															2				1	1				1		1	

Rendering Differences Arranged Tags & Properties

	BODY	article	aside	bdi	canvas	data	dd	dfn	div	dt	em	figcaption	figure	form	h1	img	ins	main	meta	meter	nav	noscript	ol	output	p	q	ruby	s	section	small	span	strong	sub	sup	tbody	td	template	tfoot	time	tr	var	video	wbr
[None]		2	1	1	2	1			28							9			3	3							1	1			12			3	1	1		3		2		19	1
animation-direction	7								3																						3						1						
animation-duration	9						1		13																						7			1									
animation-fill-mode	8								14								1														7			1			2						
animation-iteration-count	7								5							1										1			1		2	1											
animation-name				1					5									1													2												
animation-play-state	1								1																						1												
animation-timing-function	4								7																						3								1		1		
backface-visibility	1																																										
background-attachment																															1												
background-clip									1							1															1												
background-color									4																																		
border-bottom									1																																		
border-collapse									1																																		
border-style									1																					1													
border-top	1																																										
border-top-style									1										1												1												
box-shadow	1																																										
column-width									1																1																		
columns									1																																		
content	6						1		5																			1			1												
direction									3		1							1																									
display									2							1															2										1		
float			1						1						1								1																				

Figure 6.22: Reporting application TAPV view showing test results arranged by Tags & Attributes and by Tags & Property Names



Page Info

Rendering Differences Arranged Tags & Attributes

	BODY	article	aside	bdi	canvas	data	dd	dfn	div	dt	em	figcaption	figure	form	h1	img	ins	main	meta	meter	nav	noscript	ol	output	p	q	ruby	s	section	small	span	strong	sub	sup	tbody	td	template	tfoot	time	tr	var	video	wbr
[None]		2	4	2	4	3	1	1	108	1	1	1	1	1	1	3	2	2	3	2	1	1	1	1	1	2	1	3	1	1	51	1	3	7	1	1	3	2	1		1	3	1
accesskey	2								1																						1												
charset																			1	1																							
content																			1																								
contextmenu	2																																										
crossorigin																																											
dir	2																																										
height																																											
muted																																											
poster																																											
preload																																											
src																																											
tabindex	1																																										
usemap																																											
width																																											

Render

[None																																											
animation-d																																											
animation-d																																											
animation-fi																																											
animation-itera																																											
animation-																																											
animation-pl																																											
animation-timi																																											
backface-vi																																											
background-at																																											
backgroun																																											
backgroun																																											
border-bo																																											
border-col																																											
border-s																																											
border-																																											
border-top																																											
box-shad																																											
column-w																																											
columns									1																																		
content	6						1		5																			1			1												
direction									3		1							1																									
display									2							1															2										1		
float			1						1						1								1																				

X

img + src

ID	HTML	Summary	firefox	chrome	servo	Δ Avg
21	html	**Tags:** img **Attrs:** width, src **Props:** animation-iteration-count, display				
29	html	**Tags:** img, span **Attrs:** src **Props:**				
92	html	**Tags:** img, span, div **Attrs:** src, crossorigin **Props:**				
94	html	**Tags:** img, tr, div, aside **Attrs:** src, usemap **Props:**				
113	html	**Tags:** img, span **Attrs:** src **Props:**				
207	html	**Tags:** img, tfoot, div **Attrs:** src **Props:**				
211	html	**Tags:** img, tr **Attrs:** width, src **Props:**				

Figure 6.23: Reporting application TAPV view showing test cases that contain an *img* tag and contain an *src* attribute

tables. Note, that since each test may contain more than one of each feature category (tag, attribute, or property name), a test case may appear in more than one cell of each table and the total number of test case counts per table is greater (or equal) to the number of failing test cases in the data.

6.6 Bartender Usage

The concrete implementation of the Bartender approach (available at `https://github.com/kanaka/bartender`) provides a command line interface for executing the available functionality.

The examples below use the Leiningen project [36] for Clojure project management and execution. Leiningen is invoked via the `lein` command.

6.6.1 Executing Test Runs

The primary mode of execution for the Bartender system is perform one or more PBT test runs as described in Section 6.2.2. Listing 6.11 shows two command line invocations of Bartender test runs. The first command line example uses configuration entirely defined in the *config.yaml* configuration file. Appendix G contains example Bartender configuration files in the YAML format (listings G.2 and G.1) and a description of the available options (Table G.1). A second command line in Listing 6.11 shows some of the options that can override the configuration file settings including: the starting random seed value, the number of iterations per test, and the number of test runs.

```
lein run tests config.yaml
```

```
lein run tests config.yaml --seed 37 --iterations 10 --run 2
```

Listing 6.11: Execute Bartender *tests* command to start test runs

```
------
Loading http://127.0.0.1:3000/gen/28214-0-32/8.html in each browser
qc-report type: trial
------
```

```
------
Loading http://127.0.0.1:3000/gen/28214-0-32/12.html in each browser
Threshold violations: ("firefox" "chrome" "servo"), failure.
qc-report type: failure
------
```

```
------
Loading http://127.0.0.1:3000/gen/28214-0-32/102.html in each browser
Threshold violations: ("firefox" "chrome" "servo"), failure.
qc-report type: shrink-step
------
```

Listing 6.12: Bartender *tests* command output for trial, failure, and shrink iterations

Listing H.1 in Appendix H shows the initial output from Bartender when it is invoked with either of the commands described in Listing 6.11. At this point the system has loaded its configuration, started the embedded web server, initialized testing state, connected to each of the configured browser WebDriver interfaces, and is waiting for the tester to press enter to continue. The reason for this pause (which can be overridden with the *–non-interactive* option) is to give the user an opportunity to load the monitoring application to view the testing as it progresses (refer to Section 6.5).

Listing 6.12 shows output from Bartender that is reported for three different types of iterations. The first output shows a trial iteration for which there was consensus (no violations). The second output shows the initial failure iteration. The example shows that Bartender considered all three browsers in the consensus pool to be in violation. The final output shows a shrink step iteration. Once again, all three browsers are considered in violation. A more complete output example is given in Listing H.2 of Appendix H.

While Bartender is executing test runs, it generates a number of files that are stored in the datastore. For each test run, a random five digit number is created to uniquely identify the test run. This number is combined with the current test run index and the current

Table 6.2: Description of files created in the datastore during test execution. [SLUG] is the current test run slug, [IDX] is the current iteration index. [Ba] and [Bb] are the first two browser identifiers (from the configuration).

File	Description
[SLUG].edn	state serialization of all completed test runs
[SLUG]/log.edn	state serialization of one completed test run
[SLUG]/weights-start.edn	weight wtrek at start of test run
[SLUG]/weights-end.edn	weight wtrek after test run (with reduction)
[SLUG]/[IDX].html	HTML and CSS test case
[SLUG]/[IDX].html.txt	HTML and CSS test case (for raw display)
[SLUG]/[IDX]_diffs.edn	disagreement measures for every browser pair
[SLUG]/[IDX]_avg.png	avg of all browser screenshots
[SLUG]/[IDX]_avg_thumb.png	avg of all browser screenshots (thumb)
[SLUG]/[IDX]_davg.png	avg of all browser difference images
[SLUG]/[IDX]_davg_thumb.png	avg of all browser difference images (thumb)
[SLUG]/[IDX]_[Ba].png	screenshot of browser render
[SLUG]/[IDX]_[Ba]_thumb.png	screenshot of browser render (thumb)
...	(continue for all browsers)
[SLUG]/[IDX]_diff_[Ba]_[Bb].png	difference image for browser pair
[SLUG]/[IDX]_diff_[Ba]_[Bb]_thumb.png	difference image for browser pair (thumb)
...	(continue for all browser pairs)

starting pseudo-random seed number to create a test "slug". An example test slug is *58144-0-32* which indicates this is a test ID "58144", test run index 0 (the first run of the set of test runs), and starting random seed 32. For every test run, the test slug is used to created a top-level directory in the datastore. Within the test slug directory, each file generated during a quick-check iteration is prefix with the index of the iteration. The iteration index starts at 0 for the first trial iteration, increments with each trial iteration, and continues incrementing during shrinking iterations. Table 6.2 describes the files that are created during Bartender test run execution.

6.6.2 Compare Renderings of an Existing Web Page

A secondary mode of Bartender execution is using the browser consensus pool to render an existing web page test case. This is useful for comparing the rendering behavior of the browsers for an existing test case. Listing 6.13 shows how the *check-page* command can be

```
lein run check-page config.yaml test-page.html
```

Listing 6.13: Execute Bartender *check-page* command to compare consensus pool renderings

```
...
------
Loading http://127.0.0.1:3000/gen/check-page/0.html in each browser
Threshold violations: ("chrome"), failure.

-----------------------------------------------

Continuing to serve on port 3000
Press <Enter> to exit
```

Listing 6.14: Bartender *check-page* command output

invoked with a configuration file to render the *test-page.html* file. The standard Bartender configuration file format is used with this mode, but the only options that are used from the configuration are *compare*, *web*, and *browsers* (refer to Table G.1 for a definition of those options). The initial output from *check-page* is very similar to the output from the *tests* command shown in Listing H.1. Similarly, the command will pause by default before executing the comparison to give the tester an opportunity to load the monitor and then press enter. Once the tester chooses to continue, the system will load the test case and render it across the consensus pool browsers and perform the normal test case comparison process including the creation of datastore files as if for a single iteration using test slug "check-page". Listing 6.14 shows the additional output resulting the from *check-page* command. Note in this particular case the test page rendering was considered different for all three browsers in the consensus pool (all three browser identifiers appear in the violations list).

6.6.3 Parse an Existing Web Page

One of the capabilities of Instacheck is the ability to parse a grammar path frequency weight trek from an existing test case and then use that as the the active wtrek configuration. This increases the likelihood of reproducing the failures (counterexamples) that were trig-

```
lein run parse test-page.html --weights-output weights.edn
```

```
lein run parse test-page.html --weights-output weights.edn
 ↪  --html-ebnf-output html.ebnf --css-ebnf-output css.ebnf
```

Listing 6.15: Execute Bartender *parse* command to parse an existing web page test case

```
Loading HTML parser
Loading CSS parser
Processing: 'test-page.html'
  - parsing HTML
  - HTML weights: 13/1750
  - parsing CSS
  - CSS weights: 24/3524
Combined and filtered weights: 51
Generating pruned HTML EBNF
Saving pruned HTML EBNF to: 'html.ebnf'
Generating pruned CSS EBNF
Saving pruned CSS EBNF to: 'css.ebnf'
Saving merged weights to: 'weights.edn'
```

Listing 6.16: Bartender *parse* command output

gered by the existing test case. This is particular useful as a means of shrinking or simplifying an existing test case. It may also reveal other failure modes or interactions of the underlying root cause of the failure. Bartender extends this capability of Instacheck to web browser test cases. The first command in Listing 6.15 shows the basic *parse* command line to parse and store a path frequency weight trek from a web page test case. The second command shows additional options that use Instacheck to output HTML and CSS EBNF grammars that have been pruned of all unreachable paths based on the parsed path frequency trek. Listing 6.15 also shows an example of output from the second command including summaries of the weights that were parsed. The behavior and files that are generated by the *parse* command are very similar to the *html5-css3-ebnf* parse command shown in Section 5.5.2 so they are not repeated here. The primary difference is that the grammar used by Bartender has a small number of differences due to the optimizations during translation that are described in sections and 6.6.4.

```
time lein run translate --mode html --namespace rend.html5-generators
 ↪  --function html5-generators --weights-output
 ↪  resources/html5-weights.edn --clj-output
 ↪  resources/rend/html5_generators.clj
```

```
time lein run translate --mode css --namespace rend.css3-generators
 ↪  --function css3-generators --weights-output resources/css3-weights.edn
 ↪  --clj-output resources/rend/css3_generators.clj
```

Listing 6.17: Execute Bartender *translate* command to optimize and translate HTML and CSS EBNF grammars to Clojure generator code.

6.6.4 Optimize and Translate EBNF Grammars to Clojure Generators

Section 6.6.3 describes optimizations that are applied to the EBNF grammars from *html5-css3-enbf* when they are translated into Clojure generator code for use by Bartender. Listing 6.17 shows the Bartender commands for executing the translation and optimization process. The same command is invoked for both the HTML and CSS grammars but using a different mode, Clojure namespace and file paths. Weight trek files are also generated that contain the full list of weighted paths in the grammars along with their default weight values. There is no output from the command if the process is completed successfully.

Chapter 7

Overall Browser Render Testing Results

This chapter describes the results of using the Bartender system to perform browser render testing. The approach used by Bartender is introduced in Chapter 2 and described in detail in Chapter 6. The versions of the software projects, libraries, and modules that composed the Bartender system at the time these tests were performed are listed in Appendix I.

Two different sets of tests are reported here. The first set of tests was a comparison of Mozilla's Firefox browser against Google Chrome's browser. The second set of tests was a comparison of Mozilla's Servo browser against Mozilla Firefox and Google Chrome. Appendix G lists the Bartender configurations that were used for each testing mode. The following common configuration settings were used for both set of test runs:

- The number of iterations was 25.
- The maximum generator size parameter of 50.
- The normalized SSD threshold value was 0.0001.
- After each test run the weights were reduced using the *:weight* selection algorithm, the *:zero* propagation algorithm, and a 3 step reducer function (with a weight progression

of: 100, 10, 1, 0).

- The starting random seed was initialized to the current run number at the start of each new test run.

For the first set of tests, 5000 test runs were completed and for the second set 1000 test runs were completed. The test runs were executed in chunks of 100 to 500 test runs at a time. This provided the opportunity to sanity check and analyze the data as the tests proceeded and served as check-points from where the testing could be restarted in case of interruption (such as power failure).

7.1 Testing Firefox against Chrome

The first set of tests that we ran were to compare the rendering behavior of Mozilla's Firefox browser in comparison to Google's Chrome browser. The Bartender configuration that was used to perform this testing is shown in Listing G.1 of Appendix G. For this test any disagreement measure above the threshold was considered a failing test case.

In total 5000 test runs were completed that took an aggregate clock time of 89 hours (a mean of about 64 seconds per run). A total of 712,193 iterations were performed (the Bartender iteration process is described in Section 6.2.2) with each iteration taking an average of 0.45 seconds. Out of the 5000 runs, 1805 test cases resulted in detected rendering differences. After shrinking 1695 of the 1805 tests cases were unique (110 shrunk test cases were identical to another shrunk test case).

Figure 7.1 is a plot showing an overview of the 5000 test runs over time (runs are grouped into bins of 200 runs). The green line shows the percentage of test runs that are considered failures (counterexamples). The reason for the decrease over time is due to the use of the weight reduction process. Every time a failing test case is discovered, the weights are adjusted to try and reduce the probability of detecting the same test case in subsequent runs. This also has the effect of reducing the likelihood that any given test run will detect a failure

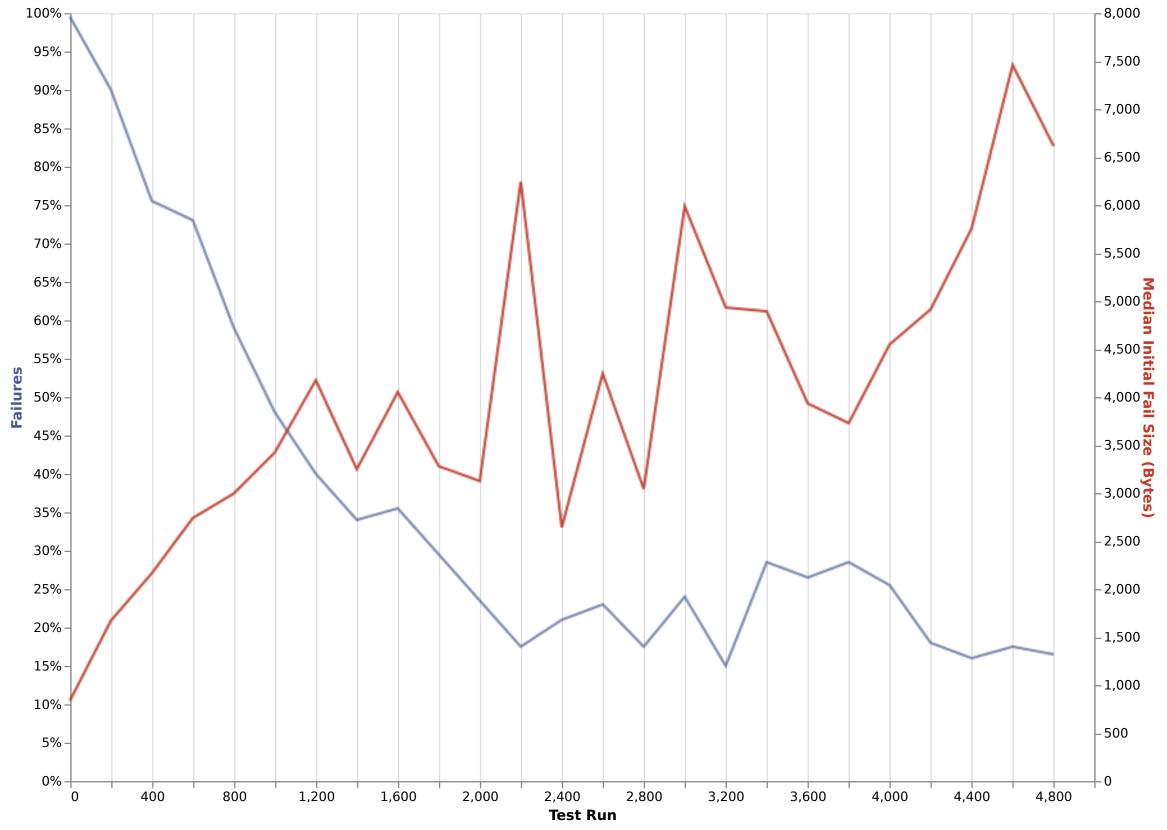


Figure 7.1: Plot of Firefox and Chrome testing runs (with weight reduction between runs) showing the median initial failure size byte count (red) and the percentage of runs per 200 runs that failed (blue)

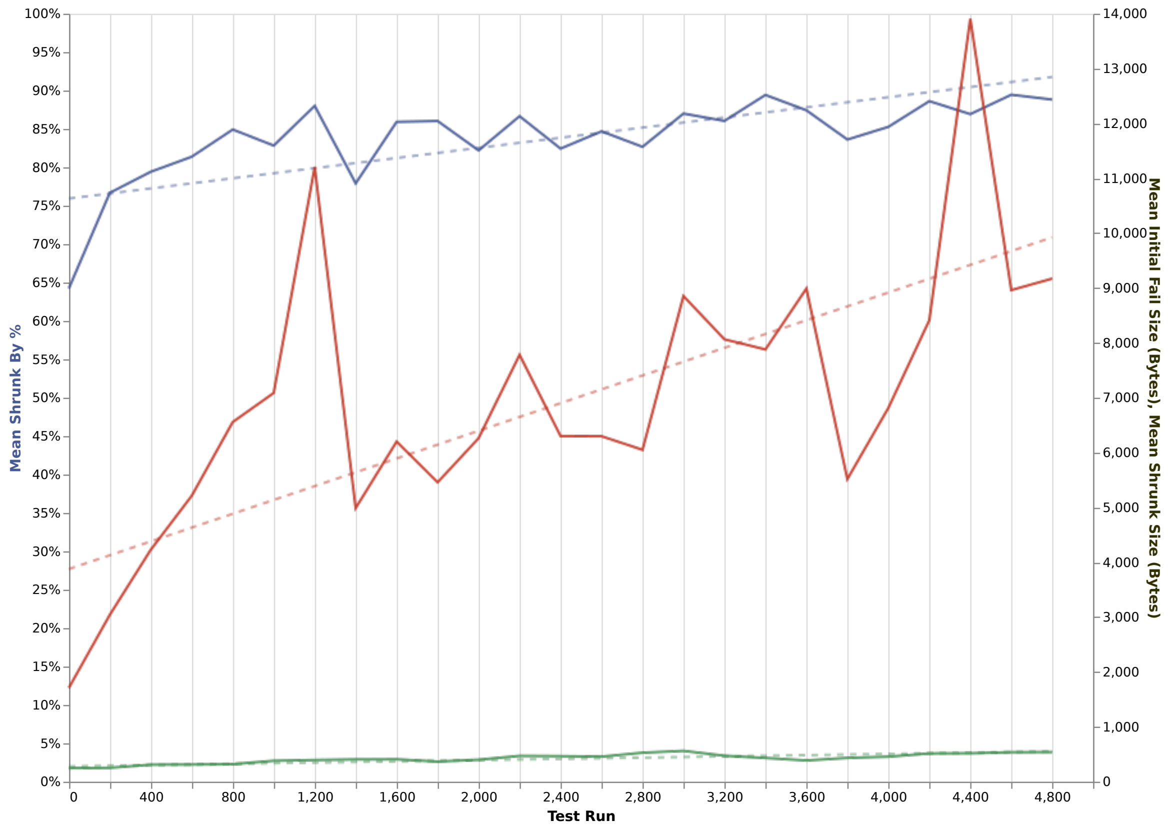


Figure 7.2: Plot of Firefox and Chrome testing runs (with reduction between runs) showing the mean initial failure size byte count (red), the mean final shrunk size byte count (green), and reduction percentage (blue). Linear regressions for each plot line are shown as dotted lines.

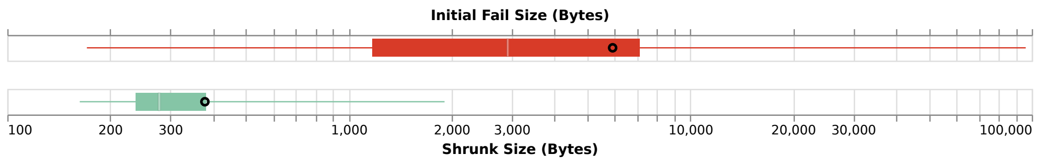


Figure 7.3: Box and whisker plots for Firefox and Chrome testing runs (with reduction between runs) showing quartiles for initial failure sizes (red) and final shrunk sizes (green). The black circles indicate the means.

before reaching the maximum of 25 trial iterations.

The blue line in Figure 7.1 shows the median byte size of the initial failing test case for runs that detect a failing test case. Initially, the test case size that is needed to detect rendering differences is fairly small (under 1000 bytes). The data shows a general increase over time in the size of test cases that are required to discover a failure (rendering counterexample). The line becomes more noisy for later runs (higher variance) because the results are binned by test run (regardless of failure result) so the later bins have smaller sample sizes (fewer test fail in later test runs).

Figure 7.1 is a plot showing an overview how well the system is able to perform shrinking. The data covers the 5000 test runs over time (runs are grouped into bins of 200 runs). The red line shows the mean byte size of the initial failure for test runs with failures. The green line shows the mean byte size of the final shrunk test case for the respective test run. The blue line shows the mean percentage decrease between the initial failure test case and the shrunk test case. Figure 7.3 shows the same data but in summary box and whisker format. The initial failing test case sizes have a minimum of 171 bytes, median size of 2921 bytes, mean of 5924 bytes, and maximum of 95993 bytes. The final shrunk test case sizes have a minimum of 163 bytes, a median size of 279 bytes, a mean of 379 bytes, and a maximum of 1907 bytes. The ratio of mean shrink size to mean initial size is 6% (a decrease of 94%). The mean of individual shrink ratios is 19% (a mean decrease of 81%).

Tables E.1, E.2, and E.3 in Appendix E enumerate the number of failing (shrunk) test cases that each HTML tag, HTML attribute, and CSS property appear in. Note that these represent a possible cause of a rendering difference between Servo as compared to Firefox and Chrome for that particular element. This does not necessarily imply that the element caused the render difference because it may be in the same test case with other elements and simply represent that the shrinking process terminated before it was able to eliminate this element. However, if the element appears in a high number of test cases then this is evidence that it may be causing or contributing to the rendering difference.

Some of the CSS property differences indicated in tables E.2, and E.3 are due to the use of animation related properties. These likely represent a race condition in the timing of when the screenshot was actually triggered in relation to the timing of the animated element. In other words, these elements might show a rendering difference even consensus pool was composed of multiple instance of the exact same browser. Using the same browser multiple times in the consensus pool may be interesting for future work as a means of detecting transient rendering differences.

7.2 Testing Servo against Firefox and Chrome

The second set of tests that we ran were to compare the rendering behavior of Mozilla's Servo browser against the combined behavior of Mozilla's Firefox browser and Google's Chrome browser.

The Bartender configuration that was used to perform this testing is shown in Listing G.2 of Appendix G. This test was configured to use the same image comparison algorithm as the Firefox/Chrome test set. However, the consensus algorithm was configured to use the target mode testing with Servo as the target. In this mode test cases are only considered failures when Servo is in disagreement with both Firefox and Chrome, but Firefox and Chrome are in agreement. The target consensus mode is described in more detail in Section 6.3.1.

In total 1000 test runs were completed that took an aggregate clock time of 60 hours (a mean of 216 seconds per run). A total of 112,073 iterations were performed (the Bartender iteration process is described in Section 6.2.2) with each iteration taking an average of 1.92

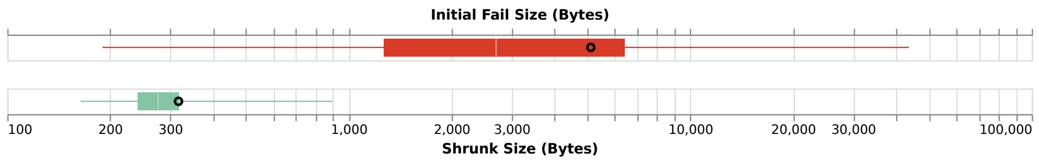


Figure 7.4: Box and whisker plots for Servo vs Firefox and Chrome testing runs (with reduction between runs) showing quartiles for initial failure sizes (red) and final shrunk sizes (green). The black circles indicate the means.

seconds. Out of the 1000 runs, 257 test cases resulted in detected rendering differences and all 257 had unique shrunk test cases.

Figure 7.4 shows box and whisker plots that summarize the shrinking process across all 1000 runs. The initial failing test case sizes have a minimum of 190 bytes, median size of 2699 bytes, mean of 5113 bytes, and maximum of 43629 bytes. The final shrunk test case sizes have a minimum of 164 bytes, a median size of 276 bytes, a mean of 317 bytes, and a maximum of 896 bytes. The ratio of mean shrink size to mean initial size is 6% (a decrease of 94%). The mean of individual shrink ratios is 18% (a mean decrease of 82%).

Tables F.1, and F.2 in Appendix F enumerate the number of failing (shrunk) test cases that each HTML tag, HTML attribute, and CSS property appear in. The same caveats apply here that applied for the Firefox/Chrome test cases in Section 7.1. In addition, tables F.3, F.4, and F.5 show a matrix representation of the result data. Each cell in Table F.3 represents and intersection of HTML tag (column) and HTML attribute (row). The count in each cell reflects the number of times that the tag and attribute combination appear in one of the 257 failing test cases. A cell with no number is equivalent to a zero count (no test case contained the combination of tag and attribute). Tables F.4 and F.5 show the same category of information but for the intersection of HTML tags and CSS properties.

A note about the data in tables F.3, F.4, and F.5: the *div* and *span* tags are excluded during weight reduction which results in those tags having abnormally high counts. The reason that *div* and *span* are excluded from the reduction process is because those tags represent the two most general grouping elements in HTML. The weights for those elements are not reduced so that they always have some chance of appearing in generated test cases. This is true even if the weights for all other HTML elements have been reduced to zero. The non-zero probability for these elements means that there is always the possibility of HTML elements to which CSS properties can be attached.

Most of the differences found between Servo and Firefox/Chrome represent differences that would be of interest to a developer that is creating web pages or web applications that should behave identically from the perspective of the end user. In other words, these difference represent modifications required to make Servo's rendering behavior match the end user perceived quality of Firefox and Chrome. Some of the differences may simply be due to implementation gaps where the functionality is incomplete and likely to be implemented in the future. However, some of the differences may also represent bugs in the current implementation. In addition to the Servo bugs listed in Table 7.1, we are also working with Mozilla to determine how we can adjust the Bartender technique to be most beneficial to Servo as development on the browser continues.

7.2.1 Bugs Discovered and Reported

Table 7.1 lists the bugs that we discovered, confirmed, and reported to the respective issue tracking systems for each browser. The *Issue ID* column contains issue numbers specific to the bug/issue tracking systems for Brave [76], Mozilla Firefox [77], and Mozilla Servo [70]. The status of the bugs in the tables and in the description below is correct as of November 2019.

Early in the testing process we discovered several issues related to use of the WebDriver interface for testing automation. In testing Firefox we found that taking a screenshot of a web page with no content would incorrectly return an empty response to the WebDriver request. The correct behavior is to return image with zero dimensions. We reported this as Firefox issue 1492357. The issue itself is not yet resolved but we were able to work around this issue during testing by generating at least a single character of content for every test case.

The Brave browser uses the same underlying rendering engine as Google Chrome [78] [79]. Although we did not perform significant testing of the Brave browser, our brief testing revealed that the WebDriver connections would timeout when the browser was launched in headless mode. We reported this as Brave issue 41523 and confirmed a fix that was provided by Brave developers.

Table 7.1: Browser bugs discovered during testing. Status correct as of 2019-10-29

Browser	**Issue ID**	**Type**	**Status**	**Description**
Firefox	1492357	WebDriver	Open	No error from WebDriver screenshot if height or width of image is 0
Brave	41523	WebDriver	Resovled	Connection timeout when using –headless with chromedriver
Servo	13825	Testing	Open	–no-native-titlebar option silently disables –headless option
Servo	13826	WebDriver	Resolved	Webdriver screenshot command timeout
Servo	16134	WebDriver	Resolved	Running headless with webdriver pegs CPU while idle
Servo	18606	WebDriver	Resolved	Headless testing (webdriver) fails to fully render on 4th load
Servo	20015	WebDriver	Resolved	Headless testing causes crash on 5th load/screenshot
Servo	23905	Core	In Progress	”Too many open files” crash after loading test cases via webdriver
Servo	23909	Core	In Progress	Introduce a shared ipc router and a ipc handle
Servo	23913	Core	In Progress	Improve IPC interfaces
Servo	23925	Core	In Progress	Use of gfx/FontSource in layout can crash
Servo	23959	Core	In Progress	Panic with Windowproxy traced while being transplanted
Servo	24042	Layout	Open	Ahem font characters render with vertical lines
Servo	24047	Core	Closed	Ensure documents and fetch cancellers drop
Servo	24052	Core	Open	Block destructor panic after window is dropped
Servo	24072	Core	Resolved	Ensure documents drop when a pipeline exits
Servo	24074	Core	In Progress	Prevent memory leak of PerformanceObserver
Servo	24088	Layout	Open	Block elem with no left/top following inline elem breaks positioning
Servo	24096	Core	Open	Ensure Task-queue doesn’t leak tasks of closed pipelines
Servo	24099	Testing	Resolved	Call gstreamer_root with top_dir string, not func
Servo	24109	Core	Resolved	Performance: limit buffer size, clear on pipeline exit

We discovered six bugs in Mozilla Servo related to either the WebDriver interface or with command line options related to automated testing. Mozilla developers have resolved four of these issues (13826, 16134, 18606, and 20015) and we have confirmed the fixes with additional testing. We proposed a fix to address issue 24099 that was accepted into the Mozilla code base. Servo issue 13825 has not yet been resolved but it is easy to avoid and does not prevent WebDriver automated testing.

We discovered eleven bugs in Mozilla Servo that caused different kinds of resource leaks that eventually resulted in a crash of the browser. Two of these (24072 and 24109) have been resolved and integrated into the main Servo source tree (master). Six issues (23905, 23909, 23913, 23925, 23959, and 24074) have proposed fixes that are not yet merged to the main Servo source tree (master). Issue 24047 has been closed in favor of the fix for 24072 which also addresses the underlying issue. Issues 24052 and 24096 are still under investigation and they may be related closely enough to some of the other issues that they will be resolved by resolutions of those other issues.

Finally, we discovered two clear Servo rendering and layout bugs that we reported that have not yet been fixed. Issue 24042 is a bug with the rendering of the Ahem font that is described in Section 6.3.2. Issue 24088 was discovered while attempting to use Bartender to reproduce and shrink an existing issue (issue 17870). Bartender reported a problem with CSS absolute positioning when a “inline” element is directly follow by a block element containing the absolute position style property.

Chapter 8

Related Work

In "Property-Based Testing of Browser Rendering Engines with a Consensus Oracle" [1] the use of property-based testing with a consensus oracle was explored as a means of fully automating the process of browser render testing. Instacheck is a refinement and formalization of the core technique with the addition of the multirun reduction capability and the ability to reproduce and shrink existing test cases. Additional related work is grouped into five categories: Property-Based Testing (PBT) / metamorphic testing, consensus oracle / differential testing, grammar-based testing, fuzz testing, and browser testing. The following sections highlight some notable work from each category of research.

8.1 Property-based Testing (PBT) / Metamorphic Testing

The term "property-based testing" was coined in "Towards a property-based testing environment with applications to security-critical software" [80] (1994) in the context of testing security-critical software. In 1997 PBT was formalized as a more general software testing technique in "Property-based testing: a new approach to testing for assurance" [81]. The basic concept is that formal program specifications are used to derive a test oracle and to generate test data.

The most popular formalization of property-based testing is the QuickCheck tool described in “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs” (2000) [10]. However, the true power of the property-based testing technique to perform test shrinking is introduced in “Testing Telecoms Software with Quviq QuickCheck” [23] (2006) and further expanded in “QuickCheck testing for fun and profit” [27] (2007).

In “Find More Bugs with QuickCheck!” [30] several techniques are proposed to address the problem of repeated discovery of the same bugs.

“RandIR: differential testing for embedded compilers” [82] (2016) describes a property-based testing system for compiler testing. The system uses a grammar syntax to generate random intermediate representation (IR) code test cases that are then differentially testing against multiple compiler implementations. The research described in “RandIR” is probably the closest related work to our research because it incorporates property-based testing, grammar-based test generators, differential / consensus oracle testing and supports testing shrinking. Apart from the obvious difference that “RandIR” is testing compiler behavior rather than browser rendering, another key differences is our system uses weighted grammars to guide generation of test cases. “Property-Based Testing with External Test-Case Generators” [83] (2017) integrates an external test-case generator into PBT to allow model-based mutation testing and give the tester more control over coverage criteria.

Metamorphic testing is a property-based testing model that defines properties as relationships between separate instances of inputs and outputs (test iterations) or between different implementations of the system (differential testing). Our research does not use metamorphic properties but this might be an interesting area to consider in the future.

Recent papers on metamorphic testing include: “Metamorphic Testing for (Graphics) Compilers” [84] (2016), “Metamorphic testing for adobe analytics data collection javascript library” [85] (2018), “Metamorphic Testing: A Review of Challenges and Opportunities” [86] (2018).

8.2 Consensus Oracle / Differential Testing

The term “test oracle” first appeared in literature in “Theoretical and empirical studies of program testing” [87] (1978). More recently, “The oracle problem in software testing: A survey” [88] (2015) give a comprehensive survey of the approaches for solving the “test oracle problem”. Section 5 describes derived test oracles which are test oracles that are derived from some type of preexisting artifact such as documentation, descriptions of properties of the system or other implementations of the system under test. A test oracle that is effectively an alternate implementation of the primary system being tested is known as a pseudo-oracle. An N-version oracle is a pseudo-oracle in which looks for a consensus or agreement across multiple implementations. In this paper we refer to this as a “consensus oracle”. The use of an N-version or consensus pseudo-oracle for testing is is often called “differential testing”.

“Differential testing for software” [19] gives an excellent introduction to differential testing with a focus on generating useful test cases. The paper covers compiler testing by using a formal grammar to generating strings from increasingly complex grammar specifications that are used to test multiple compilers to ensure they arrive at consensus result. The paper also describes using some grammar-based fuzzing techniques for performing test shrinking after the fact.

In “Crowdoracles: Can the crowd solve the oracle problem” [89] (2013), the concept of a consensus or differential test oracle is explored. However, rather than an fully automated software-based test oracle, the paper proposes using crowd sourcing techniques to get an answer from a large number of participants in the test project (for example, using Amazon’s Mechanical Turk system).

Using consensus oracle for differential testing greatly simplifies the test oracle problem for complex software. However, the challenge of generating input test cases that actually trigger bugs is still a challenge. The two main approaches to this problem are to automatically generate the inputs from some sort of specification or formal grammar or to use existing test cases and modify or fuzz them to generate new test cases. The problem with both of these methods is that without some sort of guiding function, the test process will often need very large numbers of generated tests in order to find small numbers of bugs. In “NEZHA: Efficient Domain-Independent Differential Testing” [90] (2017) this problem is addressed by introducing the notion of δ-diversity which is a measure of behavioral asymmetry between the implementations that are part of the consensus oracle. This measure is used to guide the test system towards test cases that are more likely to trigger failures. Using the NEZHA guiding algorithm to extend our system would be interesting for future work.

8.3 Grammar-based Testing

One of the earlier attempts at using formal grammars to automatically generate random test cases is described in “Generating programs from syntax” [91] 1967. “Using Attributed Grammars to Test Designs and Implementations” [92] (1981) is an discussion of improving the generation of test cases by adding additional attributes or annotations to the grammar to guiding the test generator to more interesting test cases.

“The Automatic Generation of Test Data” [93] (1987) provides a survey of approaches for automatically generating test data. In particular the paper addresses the advantages and challenges of generating test cases from syntax data (e.g. formal grammar specifications). It also covers the issue that with pure context-free grammars it is common to generate syntactically correct but semantically wrong test cases which greatly limits the code coverage that can be achieved efficiently. “Generating Test Data with Enhanced Context-Free Grammars” [94] (1990) is a readable paper that shows a case study of using enhanced context free grammars to automatically generate test data for very large-scale integration (VLSI) testing. In this case “enhanced” refers to additions to the grammar that allow the tester to guide the grammar, use variables (to related parts of the grammar) and to embed other useful information such embedded code that can generate parts of the test that may be difficult to generate with a pure context-free grammar.

"Controllable combinatorial coverage in grammar-based testing" [17] (2006) formalizes annotations that are useful for grammar based test generation and proposed a new grammar called Geno that is optimized for this type of annotation and control.

"Combinatorial Interaction Testing for Test Selection in Grammar-Based Testing" [95] (2012) applies principles of combinatorial testing to grammar-based test generation. Combinatorial testing is an approach that focuses on pairs or n-wise combinations of features of a system. Combinatorial testing has been shown to discover high numbers of system bugs while greatly limited the space of test cases that must be generated to find those bugs.

In "Combining Stochastic Grammars and GeneticProgramming for Coverage Testing at the System Level" [96] (2014) and the follow on research in "Generating Valid Grammar-based Test Inputs by means of Genetic Programming and Annotated Grammars" [97] (2017) the authors describe a system for using formal grammars to generate test cases. Two different methods are described for guiding the generation of test cases. The first paper describes the use of genetic programming with a code coverage based fitness function to tune the learned probabilities of the grammar based on a corpus corpus of existing test cases. The second paper proposes a additional approach that uses grammar annotations combined with genetic programming to guide the test generation process.

The current approach we are using uses a fairly simple form of annotated grammars where weights can be added to alternation points in the grammar. It would be interesting future work to explore more sophisticated annotations to the grammar and perhaps the use of genetic programming to increase feature coverage. It would also be interesting to use the existing annotations to implement a form of combinatorial testing.

8.4 Fuzz Testing

Unlike grammar-based test generation which randomly generate test cases from scratch, fuzz testing takes existing test cases and makes modifications to the test case to try and trigger failures in the system under test. Although the "fuzz testing" is sometimes used to refer to PBT in additional to traditional fuzz testing.

One of the problems with fuzz testing is that the effectiveness is limited when the modifications to the existing test cases are made without sufficient semantic awareness. This particularly true for highly structured inputs. "Grammar-based whitebox fuzzing" [98] (2008) proposes an enhancement to traditional fuzz testing where legal modifications are described with grammar-based constraints. This increased code coverage of Internet Explorer 7 (IE7) JavaScript interpreter from 53% to 81% while using one third the number of test cases.

Even with grammar assisted mutation as described in "Grammar-based" [98] the problem still exists that the test developer must manually create the grammar used for mutation. In "GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation" [99] (2013) and "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data" [100] (2016) this problem is addresses by having the respective systems parse a corpus of existing test data and building a grammar tree derived from that corpus which is then used to guide the mutation process. While this technique greatly improves the percentage of syntactically valid test cases that are generated and code coverage efficiency there is still a non-trivial amount of syntactically invalid test cases that are generated.

"FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage" [101] (2018) takes a different approach for increasing code coverage with fuzz testing. "FairFuzz" identifies code branches that are rarely exercised and uses this information to generate a mutation mask which is used to biased test creation towards those rare branches.

"Evaluating Fuzz Testing" [102] (2018) does an in-depth survey of 32 fuzz testing papers and identifies problems in all evaluations. The paper proposes a structured system for evaluating fuzz test system and gives guidelines for avoiding common pitfalls.

The focus of most fuzz testing is finding bugs that crash the SUT and security vulnerabilities. One reason for this is that it is difficult to create a test oracle that can correctly predict what the effect of random mutation will be on the system. For this reason the oracle is often limited to identifying crashes and security vulnerabilities. Using differential testing with fuzz testing is an area that could use more research. Another limitation of fuzz testing is that failing test cases can be large and it is difficult to shrink the test because much of the semantic information about the test case is not maintained (unlike with PBT shrinking). However the Halfempty [103] project has a method to shrink fuzz test results. Perhaps the most interesting aspect of Halfempty is that it proposes a novel approach for partitioning the shrinking process so that it can be run in parallel. This same approach would be interesting applied to the context of PBT shrinking.

8.5 Browser Testing

One of the first papers to propose a systematic approach to web application testing is "Analysis and testing of web applications" [104]. The approach described is that a web page is downloaded and analyzed to create a Unified Modelling language (UML) model of the page. This UML model is the used to generate test cases for the web application. This paper was published in 2001 which is prior to the significant growth in browser power and capabilities. The focus is on testing individual web applications with a focus on form submission behavior.

The paper "An empirical approach to evaluating web application compliance across diverse client platform configurations" [105], published in 2007, describes an approach where correctness data is collected from web applications that are running on live user browser configurations. Along with information about the HTML tags contained within a page, the system inductively determines how well a web application is expected to work given the HTML tags used in the application and the specific browser configurations.

One of the first works to directly address the problem of automating cross-browser testing is "WEBDIFF: Automated identification of cross-browser issues in web applications" [106] published in 2010. In the proposed WEBDIFF system a web application is loaded in multiple browsers and the differential testing is performed on both screenshots of the visual rendering and structural analysis of the resulting DOM. The visual comparison is done by activating and deactivating elements of the DOM and incrementally taking screenshots of the result to determine which element on the page is different. The system was used to test nine existing web applications and 121 issue were identified with 21 false positives.

In "Automated cross-browser compatibility testing" [107] (2011), web applications are tested by crawling the application to generate a finite state machine (FSM) representing the behavior of the applications. The same process is performed across multiple browser environments and any differences in the FSMs are detected.

"X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications" [108] and "Browserbite: Accurate Cross-Browser Testing via Machine Learning Over Image Features" [109] were both published in 2013 and both suggest approaches for cross-browser differential testing of web applications. The focus of "X-PERT" is apply different differencing techniques for different classes of cross-browser incompatibilities (XBIS) including structural, content and behavior. The focus of "Browserbite" is on more accurately identifying visual differences by using image segmentation in conjunction with machine learning.

In "Webpage cross-browser test from image level" [110] (2017) a new algorithm is introduced named "iterative perceptual hash" (IPH) that calculates a single difference index based on global and local color and structural differences. This approach is then used for comparing web application visual rendering within multiple browsers to find differences. The IPH approach detects attempt to imitate human nonlinear visual perception and identify global and local visual differences that are perceptible to the human eye.

The research described above is primarily focused on testing web applications to find cross-browser compatibility issues within those applications. The focus of our research is testing the browser environment itself, specifically browser rendering. Limiting the test inputs to a collection of web applications test cases would means that coverage of browser rendering functionality will be limited. In fact, if the collection of test pages is selected from published web sites, then many browser differences will have been intentionally removed by the authors of those pages.

There is significant literature related to testing of browsers with a focus on stability and security. There is also a significant number of papers on testing JavaScript engines via differential testing. However, research focusing on testing browser rendering is surprisingly limited. In other words the focus of our system is to test browser correctness across the space of possible web pages rather than testing the correctness of web applications on one or more browsers. The system we developed could be adjusted to generate test cases that would include the HTML and CSS features that are used by one or more web applications. This would automate the discovery of browser differences that have the potential to effect the rendering of those applications across different browsers.

In "Applying Model-based Testing to HTML Rendering Engines–A Case Study" [111] (2008), the authors describe a modified Behavior Adaption in Testing (BAiT) model that is used to test to correct rendering of browser CSS box models. This is done by creating a formal specification for the CSS box model and then generating simple CSS test cases based on the model. Information about the result is extracted and compared to the theoretical model. The results described in the paper were limited to testing on a single rendering engine (Mozilla Gecko) and generated small test cases. Unsurprisingly, no true positive faults were discovered.

Chapter 9

Concluding Remarks

Our research started with two initial goals. The first was to develop a testing approach that enables end-to-end automated browser render testing. Existing browser render testing techniques require manual at various stages of the testing process. The second goal was for the testing approach to be effective and practical for the users performing tests. A primary use-case for the approach was to test Mozilla's new experimental browser Servo. After discussion with a Mozilla engineer we added a third goal to overall focus: the ability to reproduce and shrink preexisting test cases.

We chose to base our the approach on the PBT model of testing where tests are defined as input properties and output properties. The input properties are used to automatically generic random test cases, and the output properties serve as a test oracle that determine if the SUT behaves correctly for a give test case. For automatic generation of test cases, we developed a module, **html5-css3-ebnf** that converts the W3C's specification data for HTML and CSS into EBNF grammars. These grammars and then translated to PBT style generators. To automate the checking of browser behavior we implemented a consensus pool oracle composed of multiple browsers from which we take screenshots of rendering results and then determine if there is a consensus or lack-of-consensus. Our initial research showed that end-to-end automation of browser render testing was possible and we published our initial results at COMPSAC in 2018 [1].

During the initial phase of research, we discovered that the standard PBT approach has some serious limitations that inhibit the testing model we were trying to develop. The first limitation is that PBT test specification are usually defined in a domain-specific language (DSL) that is closely related to the software language of the SUT rather than in a more generic test specification grammar. The second is that adjusting the coverage of those test specification typically involves manually updating the specifications or creating new specifications. The third limitation is that the normal PBT model tends to rediscover the same problems. The shrinking capability of PBT, while extremely powerful, exacerbates this problem by often shrinking new test cases to already discovered simpler test cases. Finally, PBT does not provide a direct mechanism for using the shrinking process with preexisting test cases that were not generated by the system itself.

To address the PBT issues we developed a new PBT-based approach called **Instacheck**. This approach uses EBNF grammars as a generic grammar language for test specifications to address the first issue. The remaining issues are addressed by using a grammar weighting system. The system can be used by the tester to manually adjust coverage without modifying the test specification. The system also provides automatic weight reduction algorithms that adjust the weights between test runs in order to reduce the probability of discovering failing test cases that have already been discovered. And finally, existing test cases can be parsed to generate a weight configuration that can then be used to generate similar test cases which can then be shrunk by Instacheck. The Instacheck approach is a generic testing approach and is not specific to browser render testing. We validated the Instacheck approach using a grammar that generate test cases resembling HTML and CSS web pages but is much more constrained to enable more fine grained analysis. The results show that Instacheck is able to successfully address the limitations we identified in PBT.

Bartender (**B**rowser **A**utomated **R**ender **Te**sti**N**g **D**riv**ER**) is the name of the module that leverages the capabilities of **Instacheck** and **html5-css3-ebnf** to implement the full browser render testing system. We used Bartender run over 6000 test runs testing the Firefox, Chrome, and Servo web browsers. We created reports showing over 1,500 unique test cases that trigger rendering differences between those browsers. In addition, we identified and reported over 20 confirmed bugs that we have and are continuing to work with the browser vendors to resolve. This includes a number of resource leak bugs in Servo that cause the browser to crash. We also verified that Bartender can successfully parse preexisting test cases and then reproduce and shrink those test cases.

Finally, Instacheck, html5-css3-ebnf, and Bartender fulfill our overall goal of practical utility. In addition to successfully demonstrating the ability to find bugs in real-world systems, the concrete implementations of all three modules have command line and library interfaces that are documented for ease of use by end users and developers. All three modules have implementations that have been released as open source projects. Refer to Appendix I for more information about module versions and locations. Refer to Appendix J for general statistics about the artifacts.

The Bartender approach was developed and discussed in the context of browser render testing. However, the general approach is not limited to browsers. The Bartender approach is directly applicable to other document rendering applications such as word processors, PDF viewers, image viewers, etc. In addition, the approach could be applied to any context where there are SUTs with multiple implementations that take the same structured inputs and are designed to have similar or identical behavior are candidates for the Bartender approach. Candidates include compilers, transport protocols, computer vision feature detectors, and artificial neural networks, and among others. There is existing literature exploring differential testing in a number of these contexts but Bartender and Instacheck offer some unique capabilities that could be beneficial in those domains. These capabilities include the automatic weight adjustment processes to increase coverage and reduce rediscovery; the ability to shrink tests; and the ability to parse and reproduce preexisting tests.

9.1 Future Directions

While developing the techniques described in this work, we identified a number of future directions for the work. Some of these thoughts are described below:

- The current *test.check* library that extended as part of Instacheck uses a sequential trial and shrinking process. There is significant opportunity to improve performance by adding parallel execution capability to *test.check* or switching to a PBT library that already implements parallel testing. In addition opportunities for parallel execution within Instacheck, Bartender could also be extended to perform multiple test runs in parallel. The main challenge here is that the weight reduction process would need to be adjusted to account for the parallel nature of test runs.

- There are many opportunities for extending the Instacheck and Bartender technique to work at larger scale. In particular, the support for the WebDriver standard means that Bartender can be used with web testing services running in the public cloud. Combined with parallel testing, being able to run the testing process at scale would significant increase the testing coverage that could be accomplished within a given amount time.

- The use of grammar weights in Instacheck and Bartender provides a a point of leverage for implementing additional functionality. For example, automatic pairwise (N-wise testing) testing could be implemented by selecting a pair of weights prior to each test run. The weights for other related features would be set to zero or to values much smaller than the selected pair.

- Metamorphic testing is an extension to the standard PBT testing model that would be interesting to explore in the context of Instacheck and Bartender.

- The use of *ebnf* grammars to define test generators has some limitations due to the Context-Free Grammar (CFG) nature of EBNF. In particular, there are some parts of the CSS VDS grammar that are difficult or inconvenient to represent using EBNF.

Providing an option to use PEGs grammar for the HTML and CSS grammar might provide a generations rather than EBNF.

- The WPT organization has a large repository of HTML and CSS rendering tests. These tests could be run automatically using Bartender's check-page feature to build a data set of results from different browsers. The WPT already has some tools for automated test execution, however, the benefit that Bartender could provide would be parsing those test cases to try and provide minimal versions that still trigger the differences.
- There is considerable literature on image comparison techniques. Some of these techniques would be interesting additions to the set of algorithms available in Bartender for comparing screenshot images and determining consensus.
- The Bartender approach could be used for regression testing by running different versions of the same browser in the consensus pool. The consensus algorithm and reporting functionality could be extended to have an awareness of browsers versions and order. This would enable the tester to both identify regressions and to identify when the regression was introduced (version). In addition, the ability to reproduce existing test cases would allow the tester to more efficiently and more confidently confirm when a regression is fixed.

Appendix A

Translations of *math.ebnf* Grammar to Clojure Generators

This appendix contains translations of the *math.ebnf* EBNF grammar in Listing 4.17 to Clojure code. Listings A.1 and A.2 shows the translation with each grammar rule defined as individual Clojure generators. Listings A.3 and A.4 show the translation with a single exported factory function that can be called to get the generator for a specific start rule with optional adjusted default weights.

```
(ns math.generators
  (:require [clojure.test.check.generators :as gen]
            [com.gfredericks.test.chuck.generators :as chuck]
            [instacheck.generators :as igen]

            [instacheck.util :as util]))

;; Generated by instacheck

(def gen-nz-digit
  (igen/freq :nz-digit [
    [100
      (gen/return "1")]
    [100
      (gen/return "2")]
    [100
      (gen/return "3")]
    [100
      (gen/return "4")]
    [100
      (gen/return "5")]
    [100
      (gen/return "6")]
    [100
      (gen/return "7")]
    [100
      (gen/return "8")]
    [100
      (gen/return "9")]]))

(def gen-any-digit
  (igen/freq :any-digit [
    [100
      (gen/return "0")]
    [100
      gen-nz-digit]]))

(def gen-any-number
  (igen/freq :any-number [
    [100
      gen-any-digit]
    [100
      (gen/tuple
        gen-nz-digit
        (igen/vector+
          gen-any-digit))]]))
```

Listing A.1: Translation of EBNF grammar in Listing 4.17 to Clojure generator code with one generator function per grammar rule (Part 1)

```
(def gen-expression
  (gen/recursive-gen
    (fn [inner]
      (igen/freq :expression [
        [100
          (gen/tuple
            (gen/return "(")
            inner
            (gen/return ")"))]
        [100
          (gen/tuple
            inner
            (gen/return "+")
            inner)]
        [100
          (gen/tuple
            inner
            (gen/return "-")
            inner)]
        [100
          (gen/tuple
            inner
            (gen/return "*")
            inner)]
        [100
          (gen/tuple
            inner
            (gen/return "/")
            inner)]
        [100
          gen-any-number]]))
    gen-any-number))

(def gen-line
  (gen/fmap util/flatten-text
    (gen/tuple
      gen-expression
      (igen/freq :line [
        [100
          (gen/return "")]
        [100
          (gen/return "\n")]]))))
```

Listing A.2: Translation of EBNF grammar in Listing 4.17 to Clojure generator code with one generator function per grammar rule (Part 2)

```
(ns math.generators
  (:require [clojure.test.check.generators :as gen]
            [com.gfredericks.test.chuck.generators :as chuck]
            [instacheck.generators :as igen]

            [instacheck.util :as util]))

;; Generated by instacheck

(defn- gen-math-part-0 [gmap weights]
  (let [g gmap
        w weights

        gen-nz-digit
        (igen/freq :nz-digit [
          [(get w [:nz-digit :alt 0] 100)
            (gen/return "1")]
          [(get w [:nz-digit :alt 1] 100)
            (gen/return "2")]
          [(get w [:nz-digit :alt 2] 100)
            (gen/return "3")]
          [(get w [:nz-digit :alt 3] 100)
            (gen/return "4")]
          [(get w [:nz-digit :alt 4] 100)
            (gen/return "5")]
          [(get w [:nz-digit :alt 5] 100)
            (gen/return "6")]
          [(get w [:nz-digit :alt 6] 100)
            (gen/return "7")]
          [(get w [:nz-digit :alt 7] 100)
            (gen/return "8")]
          [(get w [:nz-digit :alt 8] 100)
            (gen/return "9")]])
        g (assoc g :nz-digit gen-nz-digit)

        gen-any-digit
        (igen/freq :any-digit [
          [(get w [:any-digit :alt 0] 100)
            (gen/return "0")]
          [(get w [:any-digit :alt 1] 100)
            (:nz-digit g)]])
        g (assoc g :any-digit gen-any-digit)

        gen-any-number
        (igen/freq :any-number [
          [(get w [:any-number :alt 0] 100)
            (:any-digit g)]
          [(get w [:any-number :alt 1] 100)
            (gen/tuple
              (:nz-digit g)
              (igen/vector+
                (:any-digit g)))]])
        g (assoc g :any-number gen-any-number)
```

Listing A.3: Translation of EBNF grammar in Listing 4.17 to Clojure generator code with one generator factory function (Part 1)

```
        gen-expression
        (gen/recursive-gen
          (fn [inner]
            (igen/freq :expression [
              [(get w [:expression :alt 0] 100)
                (gen/tuple
                  (gen/return "(")
                  inner
                  (gen/return ")"))]
              [(get w [:expression :alt 1] 100)
                (gen/tuple
                  inner
                  (gen/return "+")
                  inner)]
              [(get w [:expression :alt 2] 100)
                (gen/tuple
                  inner
                  (gen/return "-")
                  inner)]
              [(get w [:expression :alt 3] 100)
                (gen/tuple
                  inner
                  (gen/return "*")
                  inner)]
              [(get w [:expression :alt 4] 100)
                (gen/tuple
                  inner
                  (gen/return "/")
                  inner)]
              [(get w [:expression :alt 5] 100)
                (:any-number g)]]))
          (:any-number g))
        g (assoc g :expression gen-expression)

        gen-line
        (gen/tuple
          (:expression g)
          (igen/freq :line [
            [(get w [:line :cat 1 :opt nil] 100)
              (gen/return "")]
            [(get w [:line :cat 1 :opt 0] 100)
              (gen/return "\n")]]))
        g (assoc g :line gen-line)]
    g))

(defn gen-math [& [gmap weights]]
  (let [g (or gmap {})
        w weights

        g (gen-math-part-0 g weights)]
    g))
```

Listing A.4: Translation of EBNF grammar in Listing 4.17 to Clojure generator code with one generator factory function (Part 2)

Appendix B

Instacheck Library Functions

This appendix summarizes the library functions that are available in the Clojure implementation of the Instacheck system (as of version 0.9.1). The Instacheck module is available at `https://github.com/kanaka/instacheck`. Tables B.1, B.2, and B.3 list and describe the functions grouped by code module.

Table B.1: Instaparse Modules and Functions Part 1

Module	**Function**	**Description**
core	grammar->generator-obj	create a generator object from a context and grammar
	update-generator-obj	update a generator object with adjusted properties
	ebnf->gen	create a generator from a EBNF string, grammar, parser, or file
	grammar->ns	create full namespace code translation for a grammar
	parse-wtrek	parse a string and return a path-log (frequency) wtrek
	parse-wtreks	parse multiple strings and return an overall path-log (frequency) wtrek
	ebnf-sample-seq	generate an infinite sequence of values generated from an EBNF string
	ebnf-generate	generate a single value generated from an EBNF string
	instacheck	run the quick-check process using an EBNF-based generator
reduce	reducer-zero	reducer that returns 0 regardless of start weight
	reducer-half	reducer that returns half the start weight
	reducer-div	reducer that returns start weight divided by divisor
	reducer-ladder	reducer that returns next weight in sequence that is lower than start-weight
	reduce-wtrek	reduce a grammar wtrek by a reduce-mode and propagate if needed
	reduce-wtrek-with-weights	reduce a grammar wtrek by a weight map and pick-mode
	prune-grammar	prune a grammar based on a wtrek and/or list of paths
	prune-grammar->sorted-ebnf	prune-grammar and return EBNF for that grammar
parse	parse	a parse function that checks for errors and has elided position reporting

Table B.2: Instaparse Modules and Functions Part 2

Module	**Function**	**Description**
grammar	parser->grammar	convert a parser object to a grammar tree
	grammar->parser	convert a grammar tree to a parser object
	load-parser	create parser object from an EBNF string
	load-grammar	create a grammar tree from an EBNF string
	grammar->ebnf	translate a grammar tree to an EBNF string
	get-in-grammar	get the grammar node at the given path in a grammar tree
	update-in-grammar	apply an update function to a given path in a grammar tree
	assoc-in-grammar	replace a node at the given path in the grammar tree
	apply-grammar-update	updates a grammar tree with an update trek
	paths-to-leaf	return paths in the grammar tree with the given non-terminal as a leaf value
	get-descendants	return descendant paths of the given path that fulfill a predicate function
	get-ancestors	return all nearest ancestors of given path that fulfills a predicate function
	get-weighted-ancestors	return nearest weighted ancestors of a given path
	children-of-node	return direct child paths of the node at the given path
	combine-strings	combine strings that are immediately adjacent in the grammar tree
	trek-grammar	walk grammar tree applying a transaction function at each path/node
	trek	return a value trek (paths to values) for a grammar tree
	comment-trek	parse grammar comments in a grammar tree as Clojure data structures
	trek->grammar	convert a value trek (paths to values) to a grammar tree

Table B.3: Instaparse Modules and Functions Part 3

Module	**Function**	**Description**
weights	removed-node?	check if a path has all zero child weights given a grammar and wtrek
	filter-trek-weighted	return weighted paths filtered from a trek
	wtrek	return a wtrek (weight trek) for a grammar tree
	path-log-trek	return path-log trek for a grammar based on a parse-result
	path-log-wtrek	return path-log wtrek for a grammar based on a parse-result
	print-weights	pretty print sorted and indented weights/wtrek
	save-weights	save sorted and indented weights/wtrek to a file
	likelihood-trek	return trek with likelihood of reaching every node of grammar based on a wtrek
	terminal-likelihood-trek	return trek of likelihood of reaching each terminal of grammar based on a wtrek
	distance-trek	return trek of distance of every node from the root of the grammar
codegen	check-and-order-rules	return dependency ordered path list from grammar (throw if indirect cycles)
	grammar->generator-defs-source	return string of Clojure generator code for a grammar; one function per rule
	grammar->generator-func-source	return string of Clojure generator code for a grammar; one factory function
	eval-generator-source	evaluate/instantiate a Clojure generator code string returning last generator
	generator-func->generator	get a named generator from a generator factory function
util	tree-matches	return a sequence of nodes in a tree data structure that match a predicate
	tree-deps	return dependency map of keys that occur under top-level keys
	remove-key	walk a tree data structure and remove all matching key/values
	tree-distances	return distances of nodes from a start node (modified Djikstra's algorithm)
	flatten-text	take tree of strings and flatten into a single string

Appendix C

Full Instaparse EBNF Syntax

Table C.1 in this appendix lists all of the Instaparse EBNF syntax features that used by *Instcheck*, *html5-css3-ebnf*, or *Bartender*. The top section of the table shows the syntax elements used by Instacheck derived grammar generators. In addition to the elements introduced in Table 3.1 of Subsection 3.4.1, this table adds the *ordered alternation* combinator (*:ord*) which is similar to the regular *alternation* combinator but when the grammar is ambiguous then the parser will prefer earlier elements of the ordered alternation. The middle section of the table lists the syntax elements that used by the CSS VDS grammar (in addition to the elements in the top section). The VDS grammar is described in Subsection 5.3.2. The bottom of the table includes the negative lookahead operator that is used by the *tags and attributes* grammar depicted in Listing D.1 of Appendix D and described in Section 5.4.

EBNF Syntax	Meaning	Key	Icon	Weight
A	non-terminal (LHS)		Ⓝ	
A	non-terminal (RHS)		Ⓝ	
"abc" *or* 'abc'	literal terminal		"	
#"abc" *or* #'abc'	regexp terminal		Ⓡ	
"" *or* '' *or* ϵ	epsilon terminal		ε	
A B *or* **A, B**	concatenation	:cat	C	
A | **B**	alternation	:alt	◇ |	✓
A / **B**	ordered alternation	:ord	◆ /	✓
A? *or* [**A**]	optional	:opt	◇ ?	✓
A⋆ *or* {**A**}	zero or more	:star	◇ *	✓
A+	one or more	:plus	▽ +	
(**A B**)	grouping			
<**A**>	hidden LHS non-terminal			
<**B**> *or* <'abc'> *etc*	hidden RHS element			
= *or* := *or* : *or* ::=	LHS / RHS delimiter			
;	rule terminator (optional)			
(⋆ comment ⋆)	comment			
!**A**	negative lookahead (PEG)			

Table C.1: List of all Instaparse EBNF syntax features used

Appendix D

Tags and Attributes Grammar

This appendix contains the simplified *tags and attributes* grammar in Listing D.1. This EBNF grammar is used for the first pass of the dual-pass parsing mode used by **html5-css3-ebnf**. The use of this grammar is described in Section 5.4.

```
html = '<!DOCTYPE html>'? S comment? elem+ ;
elem = script-elem
     / style-elem
     / svg-elem
     / ahem-elem
     / start-elem
     / end-elem
     / content ;

script-elem = '<' 'script' attrs S '>' script-data '</script>' S ;
style-elem  = '<' 'style'  attrs S '>' style-data  '</style>' S ;
svg-elem    = '<' 'svg'    attrs S '>' svg-data    '</svg>' S ;
ahem-elem   = '<' 'span'   ahem-attrs S '>' ahem-data '</span>' S ;
start-elem  = '<' tag-name attrs S '>' S;
end-elem    = '</' tag-name '>' S ;
content     = char-data
            | comment ;

<script-data> = ( #'[^<]*' | !'</script>' #'<' )+ ;
<style-data>  = ( #'[^<]*' | !'</style>' #'<' )+ ;
<svg-data>    = ( #'[^<]*' | !'</svg>' #'<' )+ ;
<ahem-data>   = ( #'.' | '&#x00c9;' ) ;

attrs = attr* ;
attr = rS attr-name S <'='> attr-val
     | rS attr-name ;
ahem-attrs = ahem-attr+ ;
ahem-attr = rS ahem-attr-name S <'='> ahem-attr-val ;
ahem-attr-name = 'class' ;
ahem-attr-val = S <'"'> 'wrap-ahem' <'"'> ;

<char-data> = #'[^<]+' ;

attr-val = S <'"'> ( #'\\.' | #'[^\\"]*' )* <'"'>
         | S <"'"> ( #"\\." | #"[^\\']*" )* <"'">
         | S #'[^"\'\s>]*' ;

<tag-name> = name ;
attr-name = name ;
<name> = #'[A-Za-z_:][A-Za-z_:\-.0-9]*' ;

<comment> = '<!--' ( #'[^-]*' | !'-->' #'-' )* '-->' ;

<S> = #'\s*' ;
<rS> = #'\s+' ;
```

Listing D.1: The *tags and attributes* grammar used for first pass of parsing by **html5-css3-ebnf**

Appendix E

Firefox versus Chrome Rendering Differences Data

This appendix summarizes data that was generated during Mozilla Firefox and Google Chrome testing. Tables E.1, E.2, and E.3 enumerate the number of failing (shrunk) test cases that each HTML tag, HTML attribute, and CSS property appear in. This data is described in detail in Section 7.1.

Table E.1: Firefox and Chrome: Number of Test Cases with Rendering Differences for each HTML Tag and HTML Attribute

HTML Tags	**Count**
img	42
abbr	31
meter	30
dd	29
q	28
b	26
video	26
keygen	25
blockquote	24
address	24
area	24
rb	23
button	22
ul	22
legend	22
table	22
audio	21
time	21
small	21
label	20
bdi	20
s	19
code	18
mark	18
menu	18
output	18
aside	17
a	17
article	17
map	17
h6	16
ins	16
canvas	16
bdo	15
del	15
option	15
h3	15
optgroup	15
caption	14
ruby	14
cite	14
em	14
fieldset	14
sub	14
dl	13
section	13
rt	13
var	13
tr	13
figure	13
object	13
embed	12
picture	12

HTML Tags	**Count**
textarea	12
nav	12
input	12
hr	12
summary	12
h2	12
rtc	12
progress	12
li	11
header	11
colgroup	11
th	11
select	11
p	11
tfoot	11
tbody	10
strong	10
u	10
dfn	10
pre	10
figcaption	10
base	10
form	10
td	10
h1	9
details	9
i	9
ol	9
main	9
datalist	9
h4	9
thead	9
sup	9
samp	9
data	8
iframe	8
kbd	8
footer	8
br	8
h5	7
source	7
noscript	7
dt	7
meta	6
wbr	5
rp	5
param	5
menuitem	4
link	4
template	3
col	3
track	2

HTML Attributes	**Count**
controls	27
border	18
width	16
crossorigin	14
src	14
contenteditable	13
sizes	12
class	11
type	10
alt	10
spellcheck	8
dir	8
translate	8
contextmenu	7
draggable	6
height	6
autofocus	6
hidden	5
title	5
href	5
dropzone	5
tabindex	4
accesskey	4
max	4
srcset	4
min	4
lang	4
value	3
form	3
download	3
muted	3
keytype	2
usemap	2
name	2
hreflang	2
id	2
label	2
ismap	2
loop	2
data	1
disabled	1
novalidate	1
rowspan	1
challenge	1
cite	1
autoplay	1
coords	1

Table E.2: Firefox and Chrome: Number of Test Cases with Rendering Differences for each CSS Property (Part 1)

CSS Properties	Count
font-family	89
animation-name	49
border-style	46
float	44
align-items	41
margin-inline-start	41
font	40
animation-iteration-count	36
position	35
all	33
background-color	33
padding	30
font-size	25
padding-block	22
margin-left	22
border-bottom-style	22
writing-mode	22
animation-play-state	22
mask-image	21
font-variant-caps	20
margin-inline	18
animation-fill-mode	18
animation-direction	18
display	17
border-bottom	17
margin-block	16
animation-timing-function	16
border-top-style	15
font-variant-position	15
column-width	15
width	15
line-height	15
margin-top	14
word-break	14
vertical-align	14
column-gap	14
block-size	14
align-content	13
border-block	13
border-block-style	12
margin	12
padding-bottom	12
overflow-wrap	12
grid-auto-rows	11
text-emphasis	11
padding-right	11
border-right-style	11
scroll-padding-inline	11
border-left	11
margin-block-start	11
text-emphasis-style	11
clip-path	10
animation-duration	10
mix-blend-mode	10
overflow-x	10
overflow	10
inset	10
page-break-inside	10
align-self	10
text-align	10
text-transform	10
counter-reset	10
border-top-left-radius	9
animation-delay	9
border-block-end-style	9
font-weight	9
padding-inline	9
border-top-width	9
overflow-y	9
scale	9
translate	9
border-block-start-style	8
grid-auto-flow	8
border-image	8
border-top	8
column-count	8
border-left-style	8
max-width	8
list-style-position	8
opacity	7
background-image	7
text-decoration-style	7
bottom	7
box-shadow	7
padding-block-start	7
background-size	7
border-block-end-color	7
margin-bottom	7
border-block-end-width	7
transition-timing-function	7
inset-block	7
text-indent	7
scroll-margin-block	7
text-orientation	7
text-align-last	7
padding-inline-start	6
table-layout	6
mask-type	6
font-synthesis	6
border-inline-end-style	6
border-radius	6
letter-spacing	6
border-inline-style	6
filter	6
content	6
unicode-bidi	6
min-height	6
border-bottom-left-radius	6
animation	6
background-origin	6
grid-template-areas	6
border	6
will-change	6
line-break	6
outline-color	6
background-repeat	6
counter-increment	6
padding-inline-end	6
flex-flow	6
transform-style	5
mask	5
list-style	5
color	5
box-decoration-break	5
font-style	5
border-spacing	5
perspective	5
font-size-adjust	5
outline-style	5
background-clip	5
word-wrap	5
padding-block-end	5
z-index	5
list-style-type	5
column-rule-color	5
margin-block-end	5
scroll-margin-block-end	5
scroll-margin-block-start	5
border-right	5
transform	5
flex-direction	5
inset-inline	5
transform-box	5
border-inline	5
margin-right	5
top	5
border-image-source	5
border-image-slice	5
scroll-margin-inline	5
border-image-repeat	5
inline-size	4
text-emphasis-position	4
padding-top	4
border-image-outset	4
max-height	4
text-shadow	4
orphans	4
border-block-start-color	4
font-variant-alternates	4
border-block-color	4
grid-column	4
min-inline-size	4
scroll-padding-block-start	4
image-orientation	4
border-end-start-radius	4
empty-cells	4
scroll-padding-bottom	4
text-decoration	4

Table E.3: Firefox and Chrome: Number of Test Cases with Rendering Differences for each CSS Property (Part 2)

CSS Properties	Count
border-bottom-width	4
scroll-margin-bottom	4
cursor	4
direction	4
border-inline-end-width	4
inset-inline-start	4
mask-clip	4
margin-inline-end	4
column-rule	4
backface-visibility	4
place-content	4
outline-width	4
shape-margin	4
height	4
columns	4
background-attachment	4
font-variant-ligatures	4
grid-auto-columns	4
mask-repeat	3
isolation	3
hyphens	3
scroll-padding	3
border-left-width	3
background	3
border-inline-color	3
tab-size	3
justify-self	3
color-adjust	3
justify-items	3
border-inline-width	3
scroll-padding-block	3
caret-color	3
object-position	3
list-style-image	3
place-self	3
border-inline-start	3
background-blend-mode	3
border-top-right-radius	3
shape-image-threshold	3
object-fit	3
column-span	3
border-right-width	3
grid-column-end	3
mask-composite	3
scroll-margin-top	3
outline	3
border-collapse	3
touch-action	3
transition-delay	3
grid	3
text-combine-upright	3
justify-content	3
scroll-padding-top	3
shape-outside	3
column-rule-style	2
border-inline-start-style	2
padding-left	2
rotate	2
flex	2
transition	2
font-variant	2
text-underline-position	2
inset-block-end	2
pointer-events	2
place-items	2
scroll-margin	2
resize	2
perspective-origin	2
scroll-padding-inline-start	2
border-inline-start-color	2
text-decoration-line	2
font-feature-settings	2
border-block-start	2
order	2
border-inline-end-color	2
grid-row-start	2
break-before	2
scroll-snap-align	2
flex-grow	2
border-block-width	2
text-overflow	2
flex-wrap	2
row-gap	2
scroll-margin-left	2
page-break-before	2
border-block-start-width	2
scrollbar-width	2
visibility	2
grid-column-start	2
text-justify	2
text-decoration-color	2
scroll-behavior	2
outline-offset	2
scroll-snap-type	2
break-inside	2
text-rendering	2
border-block-end	2
font-language-override	2
scroll-padding-inline-end	2
border-color	2
transition-property	2
border-width	2
border-inline-end	2
border-top-color	2
min-width	2
flex-basis	1
scroll-snap-stop	1
font-stretch	1
scroll-padding-right	1
caption-side	1
left	1
mask-mode	1
border-image-width	1
clear	1
scroll-padding-block-end	1
scroll-margin-inline-start	1
inset-inline-end	1
font-optical-sizing	1
grid-template-columns	1
grid-template	1
break-after	1
widows	1
mask-origin	1
box-sizing	1
border-end-end-radius	1
min-block-size	1
border-start-end-radius	1
mask-position	1
grid-area	1
border-left-color	1
hanging-punctuation	1
border-bottom-color	1
border-start-start-radius	1
grid-row-end	1
border-bottom-right-radius	1
mask-size	1
quotes	1
word-spacing	1
border-right-color	1
column-rule-width	1
white-space	1
font-kerning	1
flex-shrink	1
scroll-padding-left	1
font-variant-east-asian	1
transform-origin	1
grid-template-rows	1
text-emphasis-color	1
clip	1
inset-block-start	1
image-rendering	1
scrollbar-color	1
page-break-after	1
font-variant-numeric	1
transition-duration	1
right	1
background-position	1
scroll-margin-inline-end	1
gap	1
border-inline-start-width	1
column-fill	1
grid-row	1
scroll-margin-right	1

Appendix F

Servo versus Firefox and Chrome Rendering Differences Data

This appendix summarizes data that was generated during Mozilla Servo versus Mozilla Firefox and Google Chrome testing. Tables F.1, and F.2 in Appendix F enumerate the number of failing (shrunk) test cases that each HTML tag, HTML attribute, and CSS property appear in. Tables F.3, F.4, and F.5 show the test case data is matrix representation of the results. This data is described in detail in Section 7.2.

Table F.1: Servo vs Firefox and Chrome: Number of Test Cases with Rendering Differences for each HTML Tag and Attribute

HTML Tag	**Count**
video	20
img	12
meta	7
sup	7
aside	5
tfoot	4
canvas	4
data	3
template	3
article	3
meter	3
sub	3
s	3
tbody	2
main	2
ins	2
q	2
tr	2
bdi	2
td	2
wbr	1
h1	1
strong	1
dfn	1
nav	1
ruby	1
time	1
figcaption	1
dd	1
ol	1
section	1
noscript	1
small	1
em	1
form	1
var	1
p	1
dt	1
figure	1
output	1

HTML Attribute	**Count**
src	10
crossorigin	6
height	5
width	5
accesskey	3
poster	2
contextmenu	2
dir	2
tabindex	1
charset	1
content	1
preload	1
usemap	1
muted	1

Table F.2: Servo vs Firefox and Chrome: Number of Test Cases with Rendering Differences for each CSS Property

CSS Property	**Count**
font-size	30
animation-duration	24
animation-fill-mode	24
text-align-last	22
mask-type	17
margin-left	15
animation-timing-function	13
font	13
animation-iteration-count	13
content	11
animation-direction	11
position	11
visibility	11
font-style	10
mask-size	9
outline-color	9
perspective	7
mask-origin	7
vertical-align	7
object-position	6
animation-name	6
min-inline-size	5
page-break-inside	5
mask-position	5
outline-width	5
transition-duration	5
padding-inline-start	4
mask-mode	4
orphans	4
transition	4
display	4
background-color	4
text-align	4
padding	4
margin	4
width	4
mask-repeat	3
text-shadow	3
padding-block-end	3
direction	3
z-index	3
object-fit	3
text-indent	3
word-break	3
margin-inline-end	3
max-width	3
border-top-style	2
left	2
opacity	2
padding-left	2
pointer-events	2
perspective-origin	2
word-wrap	2
margin-top	2
page-break-before	2
quotes	2
animation-play-state	2
column-width	2
padding-bottom	2
min-width	2
float	2
font-stretch	1
mix-blend-mode	1
place-items	1
scroll-margin-inline-start	1
padding-right	1
box-shadow	1
min-block-size	1
order	1
background-clip	1
font-weight	1
margin-bottom	1
min-height	1
border-top	1
row-gap	1
border-style	1
mask-composite	1
scroll-margin-block-start	1
border-collapse	1
outline-offset	1
margin-right	1
backface-visibility	1
inset-block-start	1
scale	1
page-break-after	1
columns	1
top	1
background-attachment	1
right	1
padding-inline-end	1
border-bottom	1

Table F.3: Servo vs Firefox and Chrome: HTML Tags and HTML Attributes Matrix

	BODY	article	aside	bdi	canvas	data	dd	dfn	dt	em	figcaption	figure	form	h1	img	ins	main	meta	meter	nav	noscript	ol	output	p	q	ruby	s	section	small	strong	sub	sup	tbody	td	template	tfoot	time	tr	var	video	wbr
[None]		2	4	2	4	3	1	1	1	1	1	1	1	1	3	2	2	3	2	1	1	1	1	1	2	1	3	1	1	1	3	7	1	1	3	2	1		1	3	1
accesskey	2																																								
charset																		1	1																						
content																		1																							
contextmenu	2																																								
crossorigin															2			1																1						4	
dir	2																																								
height		1																																						5	
muted																		1																						1	
poster																																								2	
preload																																				1				1	
src			1												7																					1		2		3	
tabindex	1																																								
usemap			1												1																							1			
width															4																		1	1				1		1	

Table F.4: Servo vs Firefox and Chrome: HTML Tags and CSS Properties Matrix Part 1

	BODY	article	aside	bdi	canvas	data	dd	dfn	dt	em	form	h1	img	ins	main	meta	meter	ol	output	p	q	ruby	s	section	small	strong	sub	sup	tbody	td	template	tfoot	time	tr	var	video	wbr
[None]		2	1	1	2	1							9			3	3					1	1					3	1	1		3		2		19	1
animation-direction	7																														1						
animation-duration	9						1																					1									
animation-fill-mode	8													1														1			2						
animation-iteration-count	7												1								1			1		1											
animation-name				1											1																						
animation-play-state	1																																				
animation-timing-function	4																																1		1		
backface-visibility	1																																				
background-attachment																																					
background-clip													1																								
background-color																																					
border-bottom																																					
border-collapse																																					
border-style																									1												
border-top	1																																				
border-top-style																1																					
box-shadow	1																																				
column-width																				1																	
columns																																					
content	6						1																1														
direction										1					1																						
display													1																						1		
float			1									1						1																			
font	11																																				
font-size	23		1											1														1			1						
font-stretch																															1						
font-style	7															1																					
font-weight	1																																				
inset-block-start	1																																				
left																											1										
margin	1															1							1														
margin-bottom														1																							
margin-inline-end																							1														
margin-left	6			1											1																						
margin-right																																					
margin-top	1																																				
mask-composite	1																																				
mask-mode	2																																				
mask-origin	4																																				
mask-position	2				1			1								1																1					
mask-repeat	1															1			1																		
mask-size	5				1											1												1									
mask-type	5	1							1		1					1							1					1		1							
max-width	1																																				
min-block-size											1																										

Table F.5: Servo vs Firefox and Chrome: HTML Tags and CSS Properties Matrix Part 2

	BODY	aside	data	dfn	dt	em	figcaption	figure	form	h1	img	ins	main	meta	nav	noscript	ol	output	p	q	section	small	strong	sub	tbody	td	template	time	video
min-height	1																												
min-inline-size	3															1													
min-width	1											1																	
mix-blend-mode																													
object-fit	1																												1
object-position	3	1	1								1														1				
opacity																										1			
order	1																												
orphans	4																												
outline-color	4					1				1		1							1										
outline-offset	1																												
outline-width	4																												
padding	2																												
padding-block-end																											1		
padding-bottom				1																	1								
padding-inline-end																													
padding-inline-start	2																							1					
padding-left																													
padding-right																							1						
page-break-after	1																												
page-break-before	1																												
page-break-inside	3	2															1												
perspective	3	1																		1									
perspective-origin								1							1							1							
place-items																						1							
pointer-events	2																												
position	3		1		1				1									1											
quotes																				2									
right									1																				
row-gap																						1							
scale																						1							
scroll-margin-block-start																						1							
scroll-margin-inline-start																						1							
text-align	2																										1		
text-align-last	12							1							1	1											1		
text-indent																													
text-shadow	3																												
top	1																												
transition	4																												
transition-duration	4																												
vertical-align																											1		
visibility	3																												
width	1	1									1		1	1											1				
word-break			1				1																					1	
word-wrap		1																											
z-index	1																	1						1					

Appendix G

Bartender Configurations used for Render Testing

This appendix contains the Bartender configuration files that are used for Mozilla Firefox and Google Chrome testing (Section 7.1) and for Mozilla Servo versus Firefox and Chrome testing (Section 7.2). Listing G.1 contains the Firefox and Chrome test configuration and Listing G.2 contains the Servo vs Firefox and Chrome test configuration. Listing contains a part of configuration that enables the use of the BrowserStack testing service. The various elements that make up a Bartender configuration file are described in Table G.1. The usage of configuration files in Bartender is described in Section 6.6.

```
runs: 500
reduce-weights:
    pick-mode: weight  # weight, dist, or weight-dist
    reduce-mode: zero  # zero, max-child, or reducer
    reducer-div: 10
start-seed: 0
quick-check:
    iterations: 25
    max-size: 50
weights:
    base: ["resources/html5-weights.edn",
           "resources/css3-weights.edn",
           "resources/default-weights.edn"]
compare:
    method: "SQDIFF_NORMED"  # SQDIFF
    threshold: 0.0001
web:
    host: "127.0.0.1"
    port: 3000
    dir: "gen"
browsers:
    firefox:
        url: "http://localhost:7000"
        capabilities: {"moz:firefoxOptions":
                        {"args": ["--headless"]}}
    chrome:
        url: "http://localhost:7001"
        capabilities: {"chromeOptions":
                        {"args": ["--headless"]}}
```

Listing G.1: Bartender Configuration used for Testing Firefox against Chrome

```
runs: 100
reduce-weights:
    pick-mode: weight  # weight, dist, or weight-dist
    reduce-mode: zero  # zero, max-child, or reducer
    reducer-div: 10
start-seed: 0
quick-check:
    iterations: 25
    max-size: 50
weights:
    base: ["resources/html5-weights.edn",
           "resources/css3-weights.edn",
           "resources/default-weights.edn"
           "firefox-chrome-weights.edn"]
compare:
    method: "SQDIFF_NORMED"  # SQDIFF
    threshold: 0.0001
    target: servo
web:
    host: "127.0.0.1"
    port: 3000
    dir: "gen"
browsers:
    firefox:
        url: "http://localhost:7000"
        capabilities: {"moz:firefoxOptions":
                       {"args": ["--headless"]}}
    chrome:
        url: "http://localhost:7001"
        capabilities: {"chromeOptions":
                       {"args": ["--headless"]}}
    servo:
        url: "http://192.168.88.2:7002"
```

Listing G.2: Bartender Configuration used for Testing Servo against Firefox and Chrome

```
...
browsers:
    bs-chrome-win-62:
        url: "https://USER:KEY@hub-cloud.browserstack.com/wd/hub"
        capabilities: {"browserstack.local": true,
                       "browser": "Chrome",
                       "browser_version": "69.0",
                       "os": "Windows",
                       "os_version": "10",
                       "resolution": "1024x768"}
    bs-firefox-win-62:
        url: "https://USER:KEY@hub-cloud.browserstack.com/wd/hub"
        capabilities: {"browserstack.local": true,
                       "browser": "Firefox",
                       "browser_version": "62.0",
                       "os": "Windows",
                       "os_version": "10",
                       "resolution": "1024x768"}
    bs-edge-win-17:
        url: "https://USER:KEY@hub-cloud.browserstack.com/wd/hub"
        capabilities: {"browserstack.local": true,
                       "browser": "Edge",
                       "browser_version": "17.0",
                       "os": "Windows",
                       "os_version": "10",
                       "resolution": "1024x768"}
```

Listing G.3: Portion of Bartender configuration used for testing with BrowserStack

Table G.1: Description of Bartender configuration file options

Path	**Type**	**Description**
runs	integer	total number of test runs to execute
start-seed	integer	pseudo-random number generator seed start
quick-check	CONFIG	quick-check execution config
quick-check ⇒ iterations	integer	maximum trial iterations per test run
quick-check ⇒ max-size	integer	maximum size to use with generator functions
compare	CONFIG	browser consensus config
compare ⇒ method	string	algorithm for measuring disagreement
compare ⇒ threshold	float	image disagreement measure threshold value
reduce-weights	CONFIG	optional weight reduction config
reduce-weights ⇒ pick-mode	string	selection algorithm for weight reduction
reduce-weights ⇒ reduce-mode	string	propagation algorithm for weight reduction
reduce-weights ⇒ reducer-div	integer	individual weight divisor for weight reduction
weights	CONFIG	initial weight/wtrek config
weights ⇒ base	list	wtrek files to merge into default wtrek config
weights ⇒ start	string	starting wtrek config (others set to zero)
web	CONFIG	embedded web server config
web ⇒ host	string	optional IP to bind embedded web service
web ⇒ port	integer	port to bind embedded web service to
web ⇒ dir	string	path to store test cases, images, logs
browsers	CONFIG	browsers consensus pool config
browsers ⇒ NAME	CONFIG	browser config (NAME is arbitrary string)
browsers ⇒ NAME ⇒ url	string	URL of WebDriver port for the browser
browsers ⇒ NAME ⇒ capabilities	CONFIG	Browser capabilities requested

Appendix H

Bartender Example Test Run

This appendix contains the output from the execution of a Bartender test run (*tests* command). Listing H.2 has elided lines (shown with "...") and the large test case HTML has been elided (shown with "............").

```
HTTP kit server started on port: 3000, serving directory: gen
Loading HTML parser
Loading CSS parser
Initializing browser session for: :firefox
Nov 01, 2019 9:00:01 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
Initializing browser session for: :chrome
Nov 01, 2019 9:00:02 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Initializing browser session for: :servo
Nov 01, 2019 9:00:02 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
Test Mode: tests
Test Configuration:
{:quick-check {:iterations 25, :max-size 100},
 :weights {:base :ELIDED-5243, :start :ELIDED-0},
 :verbose 0,
 :browsers
 {:firefox
  {:url "http://localhost:7000",
   :capabilities {:moz:firefoxOptions {:args ("--headless")}}},
  :chrome
  {:url "http://localhost:7001",
   :capabilities {:chromeOptions {:args ("--headless")}}},
  :servo {:url "http://192.168.88.2:7002"}},
 :compare {:method "SQDIFF_NORMED", :threshold 1.0E-4},
 :start-seed 32,
 :runs 10,
 :reduce-weights
 {:pick-mode "weight", :reduce-mode "zero", :reducer-div 10},
 :web {:host "127.0.0.1", :port 3000, :dir "gen"}}

Press <Enter> to start tests
```

Listing H.1: Bartender *tests* command output (Part 1)

```
...
-------------------------------------------------------------
------ Run index 1 --------------------
Test State:
...
------
Loading http://127.0.0.1:3000/gen/28214-0-32/0.html in each browser
qc-report type: trial
------
Loading http://127.0.0.1:3000/gen/28214-0-32/1.html in each browser
qc-report type: trial
------
...
------
Loading http://127.0.0.1:3000/gen/28214-0-32/11.html in each browser
qc-report type: trial
------
Loading http://127.0.0.1:3000/gen/28214-0-32/12.html in each browser
Threshold violations: ("firefox" "chrome" "servo"), failure.
qc-report type: failure
------
...
------
Loading http://127.0.0.1:3000/gen/28214-0-32/115.html in each browser
Threshold violations: ("firefox" "chrome" "servo"), failure.
qc-report type: shrink-step
qc-report type: shrunk
------
Quick check results:
{:shrunk
 {:total-nodes-visited 54,
  :depth 14,
  :pass? false,
  :result false,
  :result-data nil,
  :time-shrinking-ms 198959,
  :smallest
  ["<!DOCTYPE html><html><head><link rel=\"stylesheet\" href=\"/static/normalize.css\"><link
  ↪  rel=\"stylesheet\" href=\"/static/rend.css\"></head><body>x<button></button><div
  ↪  dir=\"\"></div></body></html>"]},
 :failed-after-ms 24213,
 :start-time #inst "2019-11-02T05:01:46.485-00:00",
 :num-tests 13,
 :end-time #inst "2019-11-02T05:05:29.746-00:00",
 :seed 32,
 :fail
 ["<!DOCTYPE html><html>...........</html>"],
 :result false,
 :result-data nil,
 :failing-size 12,
 :pass? false,
 :elapsed-ms 223261,
 :current-seed 32}
------
Final test state: gen/28214.edn

-----------------------------------------------

Continuing to serve on port 3000
Press <Enter> to exit
```

Listing H.2: Bartender *tests* command output (Part 2)

Appendix I

Project/Library Versions and Locations

This appendix provides a list of the projects and libraries that were created or used as part of this research. Table I.1 lists the modules that were either created from scratch or modified in significant ways. The source location, version, git source control hash, and code license are listed for each module. Table I.2 lists the primary dependencies of the Bartender, Instacheck, and html5-css3-ebnf modules. Each dependency listed with its common name, version, short description, and source repository page. Table I.3 lists projects and libraries that were either used during runtime testing (such as browsers) or during data analysis and visualization. Each is listed with its common name, version, short description, and source repository or project page.

Table I.1: Projects/Modules Created or Modified to support this research

Module	Version	git hash	Project Site	License
Instacheck	0.9.1	c5cb532	https://github.com/kanaka/instacheck	MPL 2.0 [112]
html5-css3-ebnf	0.6.4	b7057b1	https://github.com/kanaka/html5-css3-ebnf	MPL 2.0 [112]
Bartender	0.3.6	4357fde	https://github.com/kanaka/bartender	MPL 2.0 [112]
Instaparse (modified)	1.4.9.3	410e2b8	https://github.com/kanaka/instaparse	EPL 1.0 [113]

Table I.2: Projects/Libraries used by Instacheck, html5-css3-ebnf, and Bartender

Name	Version	Description	Project Site
Clojure	1.10.0	Clojure programming language	https://github.com/clojure/clojure
ClojureScript	1.10.520	Clojure to JavaScript compiler	https://github.com/clojure/clojurescript
test.check	0.10.0-alpha3	QuickCheck for Clojure	https://github.com/Clojure/test.check
test.chuck	0.10.0-alpha3	Utility library for test.check	https://github.com/gfredericks/test.chuck
Instaparse	1.4.9	Clojure context-free grammar parsers	https://github.com/Engelberg/instaparse
Differ	0.3.2	Diffing and patching Clojure data	https://github.com/Skinney/differ
Specter	1.1.0	Query/transform recursive Clojure data	https://github.com/redplanetlabs/specter
Hickory	0.7.0	Translate HTML to/from Clojure data	https://github.com/davidsantiago/hickory
Transit-clj	0.8.319	Clojure data marshalling/transport	https://github.com/cognitect/transit-clj
Transit-cljs	0.8.256	Transit for ClojureScript	https://github.com/cognitect/transit-cljs
ring	1.7.1	Modular HTTP server abstraction	https://github.com/ring-clojure/ring
OpenCV	2.4.9	Open Source Computer Vision Library	https://github.com/opencv/opencv
Selenium	3.141.59	WebDriver Browser automation	https://github.com/SeleniumHQ/selenium
Reagent	0.8.1	ClojureScript interface to React.js	https://github.com/reagent-project/reagent
Antizer	0.3.1	ClojureScript Ant React Components	https://github.com/priornix/antizer
Meander	0.0.137	Clojure data transformation	https://github.com/noprompt/meander
Oz	1.6.0-alpha6	Clojure data viz. with Vega/Vega-Lite	https://github.com/metasoarous/oz
Vega	5.4.0	Visualization grammar	https://github.com/vega/vega
Vega-Lite	4.0.0-beta.0	Grammar of interactive graphics	https://github.com/vega/vega-lite

Table I.3: Projects/Libraries used by for test runs, data analysis, and data visualization

Name	Version	Description	Project Site
Chrome	75.0.3770.142	Google Chrome web browser	https://www.google.com/chrome/
Firefox	68.0	Mozilla Firefox web browser	https://hg.mozilla.org/mozilla-central/
Servo	git 9451a00	Mozilla Servo web browser	https://github.com/servo/servo
Meander	0.0.137	Clojure data transformation	https://github.com/noprompt/meander
Oz	1.6.0-alpha6	Clojure data viz. with Vega/Vega-Lite	https://github.com/metasoarous/oz
Vega	5.4.0	Visualization grammar	https://github.com/vega/vega
Vega-Lite	4.0.0-beta.0	Grammar of interactive graphics	https://github.com/vega/vega-lite

Appendix J

Artifact Statistics

This appendix provides some summary statistics about the artifacts that were created or generated as part of the research work. During the two phases of browser testing described in 7, Bartender generated over 12 Gigabytes of test data composed of 13.5 million individual files. Tables J.1 and J.2 list the number of lines of code in the three modules that make up the test system. The monitoring and reporting application in Bartender are listed separately from the Bartender core code. In addition, the line count include code that defines 230 unit tests.

Component	**Clojure & ClojureScript**	**EBNF**	**HTML & CSS**	**Total**
Instacheck	3,514			3,514
html5-css3-ebnf	1,281	325		1,606
Bartender Core	1,330	22	208	1,560
Bartender Web Apps	1,011		414	1,425
Total	7,136	347	622	8,105

Table J.1: Written lines of code excluding comments and blank lines (SLOC)

Component	**Clojure & ClojureScript**	**EBNF**	**CSS VDS**	**Total**
html5-css3-ebnf		8,163	634	8,797
Bartender Core	21,586			21,586
Total	21,586	8,163	634	30,383

Table J.2: Generated lines of code excluding comments and blank lines (SLOC)

Bibliography

[1] J. Martin and D. Levine, "Property-based testing of browser rendering engines with a consensus oracle," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2018, pp. 424–429.

[2] A. Eicholz, S. Faulkner, S. Moon, T. Leithead, and A. Danilo, "HTML 5.2," W3C, W3C Recommendation, Dec. 2017, https://www.w3.org/TR/2017/REC-html52-20171214/.

[3] F. Rivoal, T. A. Jr., and E. Etemad, "CSS snapshot 2018," W3C, W3C Note, Jan. 2019, https://www.w3.org/TR/2019/NOTE-css-2018-20190122/.

[4] I. Hickson. (2008, Dec.) Acid tests. http://www.acidtests.org/.

[5] First commit to servo browser engine. https://github.com/servo/servo/commit/ce30d4520d67f2c6ef960571a9b3e450c5dcbebe.

[6] Servo Developers. (2017, Sep.) Servo design wiki page. https://github.com/servo/servo/wiki/Design/5aa3f8c3480e98cf2b72df470e2e333825046954.

[7] Rust Language. https://rust-lang.org/.

[8] Servo Developers. (2017, Sep.) Servo design wiki page. https://github.com/servo/servo/wiki/Design/5aa3f8c3480e98cf2b72df470e2e333825046954#the-task-architecture.

[9] T. A. Jr., F. Rivoal, and E. Etemad, “CSS snapshot 2015,” W3C, WD not longer in development, Oct. 2015, http://www.w3.org/TR/2015/NOTE-css-2015-20151013/.

[10] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” 2000.

[11] R. Hickey and other authors. Clojure. [Online]. Available: https://clojure.org/

[12] R. Draper and G. Fredericks. test.check. [Online]. Available: https://github.com/clojure/test.check

[13] S. Stewart and D. Burns, “Webdriver,” W3C, W3C Recommendation, Jun. 2018, https://www.w3.org/TR/2018/REC-webdriver1-20180605/.

[14] M. Engelberg. Instaparse. [Online]. Available: https://github.com/Engelberg/instaparse

[15] Mozilla and individual contributors. (2018, Jan.) CSS3. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS3

[16] E. Etemad and T. A. Jr., “CSS values and units module level 3,” W3C, Candidate Recommendation, Jan. 2019, https://www.w3.org/TR/2019/CR-css-values-3-20190131/.

[17] R. Lämmel and W. Schulte, “Controllable combinatorial coverage in grammar-based testing,” in *IFIP International Conference on Testing of Communicating Systems*. Springer, 2006, pp. 19–38.

[18] D. R. Kuhn, R. N. Kacker, and Y. Lei, “Sp 800-142. practical combinatorial testing,” 2010.

[19] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[20] L. Lazaris. (2011, Sep.) How do browsers render the different css border style values? [Online]. Available: https://www.impressivewebs.com/comparison-css-border-style/

[21] T. Fahrner, P. Nelson, and S. Malkin. (2016, Jul.) Ahem font. [Online]. Available: https://www.w3.org/Style/CSS/Test/Fonts/Ahem/

[22] W. Authors. (2018, Jan.) The ahem font. http://web-platform-tests.org/writing-tests/ahem.html.

[23] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2006, pp. 2–10.

[24] J. Hughes, "How to specify it!" 2019.

[25] S. Sahayam. (2017, Apr) How does scalacheck shrinking work? [Online]. Available: https://blog.ssanj.net/posts/2017-04-12-how-does-scalacheck-shrinking-work.html

[26] N. Smallbone. (2016, Dec) Different approach to shrinking (comment). [Online]. Available: https://github.com/nick8325/quickcheck/issues/130#issuecomment-265606351

[27] J. Hughes, "Quickcheck testing for fun and profit," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007, pp. 1–32.

[28] L. M. Castro, P. Lamela, and S. Thompson, "Making property-based testing easier to read for humans," *Computing and Informatics*, vol. 35, no. 4, pp. 890–913, 2017.

[29] Wikipedia contributors, "Observable universe," 2019, [Online; accessed 2019-11-06]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Observable_universe&oldid=924843744

[30] J. Hughes, U. Norell, N. Smallbone, and T. Arts, "Find more bugs with quickcheck!" in *Proceedings of the 11th International Workshop on Automation of Software Test*. ACM, 2016, pp. 71–77.

[31] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[32] World Wide Web Consortium (W3C). [Online]. Available: https://www.w3.org/

[33] J. Nelson, "The design and use of quickcheck," 2017, [Online; accessed 2019-11-01]. [Online]. Available: https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html

[34] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369025

[35] T. Ormandy. (2013, Aug.) Kahn's topological sort in clojure. [Online]. Available: https://gist.github.com/alandipert/1263783/11de850ddeff201330fbd1f78d2518c451fa509b

[36] P. Hagelberg. Leiningen. [Online]. Available: https://github.com/technomancy/leiningen

[37] CERN. http://home.cern.

[38] WHATWG. https://whatwg.org/.

[39] HTML5 History. https://dev.w3.org/html5/spec-LC/introduction.html#history.

[40] W3C Authors. (2018, Jan.) HTML. https://github.com/w3c/html.

[41] D. Santiago. Hickory. [Online]. Available: https://github.com/davidsantiago/hickory

[42] S. Ruby and A. van Kesteren, "URL," W3C, WD not longer in development, Dec. 2016, https://www.w3.org/TR/2016/NOTE-url-1-20161206/.

[43] B. Bos. (2019) CSS Specifications. [Online]. Available: https://www.w3.org/Style/CSS/current-work

[44] Mozilla and individual contributors. (2018, Jan.) Value definition syntax. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS/Value_definition_syntax

[45] R. Weakley, "Understanding the css property value syntax," 2016, [Online; accessed 2019-11-01]. [Online]. Available: https://www.smashingmagazine.com/2016/05/understanding-the-css-property-value-syntax/

[46] M. Team. (2018, Jan.) MDN Web Technology Data. https://github.com/mdn/data.

[47] S. Chacon and B. Straub, "Pro git (second edition)," 2014, [Online; accessed 2019-11-01]. [Online]. Available: https://git-scm.com/book/en/v2/Git-Tools-Submodules

[48] Web Standards Project. https://www.webstandards.org.

[49] Wikipedia contributors, "Web standards project," 2019, [Online; accessed 2019-11-19]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Web_Standards_Project&oldid=926378476

[50] Acid1 test. https://www.w3.org/Style/CSS/Test/CSS1/current/test5526c.htm.

[51] Wikipedia contributors, "Acid1," 2019, [Online; accessed 2019-11-19]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Acid1&oldid=926881280

[52] Acid2 test. http://acid2.acidtests.org/#top.

[53] Wikipedia contributors, "Acid2," 2019, [Online; accessed 2019-11-19]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Acid2&oldid=926717324

[54] Acid3 test. http://acid3.acidtests.org/.

[55] Wikipedia contributors, "Acid3," 2019, [Online; accessed 2019-11-19]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Acid3&oldid=924197884

[56] WPT Authors. (2019) web-platform-tests. https://web-platform-tests.org.

[57] P. le Hegaret. (2017, May) The web-platform-tests project. [Online]. Available: https://www.w3.org/blog/2017/05/the-web-platform-tests-project/

[58] WPT Authors. (2019) Writing Tests. https://web-platform-tests.org/writing-tests/index.html.

[59] ——. (2019) Reftests. https://web-platform-tests.org/writing-tests/reftests.html.

[60] ——. (2019) Visual Tests. https://web-platform-tests.org/writing-tests/visual.html.

[61] R. Hickey and other authors. Clojurescript. [Online]. Available: https://github.com/clojure/clojurescript

[62] Wikipedia contributors, "Bug compatibility," 2019, [Online; accessed 2019-11-10]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bug_compatibility&oldid=916996650

[63] OpenCV. https://github.com/opencv/opencv.

[64] OpenCV Template Matching. https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html.

[65] Wikipedia contributors, "Quirks mode," 2019, [Online; accessed 2019-11-10]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Quirks_mode&oldid=923879006

[66] W3C Authors. (2019) Doctypes and markup styles. https://www.w3.org/wiki/Doctypes_and_markup_styles.

[67] "What is a css reset?" 2019, [Online; accessed 2019-11-01]. [Online]. Available: https://cssreset.com/what-is-a-css-reset/

[68] E. A. Meyer, "Css tools: Reset css," 2019, [Online; accessed 2019-11-01]. [Online]. Available: https://meyerweb.com/eric/tools/css/reset/

[69] Nicolas Gallagher. A modern alternative to CSS resets. https://github.com/necolas/normalize.css.

[70] Servo Issues. https://github.com/servo/servo/issues/.

[71] Selenium. https://github.com/SeleniumHQ/selenium.

[72] BrowserStack. https://www.browserstack.com.

[73] Reagent. https://github.com/reagent-project/reagent.

[74] React. https://reactjs.org.

[75] Differ. https://github.com/Skinney/differ.

[76] Brave Community / Issues. https://community.brave.com/t/.

[77] Firefox Issues. https://bugzilla.mozilla.org/show_bug.cgi.

[78] Brave authors. brave-browser/package.json. https://github.com/brave/brave-browser/blob/dd0ad3450/package.json#L32.

[79] Wikipedia contributors, "Brave (web browser)," 2019, [Online; accessed 2019-11-10]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Brave_(web_browser)&oldid=926442562

[80] G. Fink, C. Ko, M. Archer, and K. Levitt, "Towards a property-based testing environment with applications to security-critical software," CALIFORNIA UNIV DAVIS DEPT OF COMPUTER SCIENCE, Tech. Rep., 1994.

[81] G. Fink and M. Bishop, "Property-based testing: a new approach to testing for assurance," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, pp. 74–80, 1997.

[82] G. Ofenbeck, T. Rompf, and M. Püschel, "Randir: differential testing for embedded compilers," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. ACM, 2016, pp. 21–30.

[83] B. K. Aichernig, S. Marcovic, and R. Schumi, “Property-based testing with external test-case generators,” in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 337–346.

[84] A. F. Donaldson and A. Lascu, “Metamorphic testing for (graphics) compilers,” in *Proceedings of the 1st International Workshop on Metamorphic Testing*. ACM, 2016, pp. 44–47.

[85] Z. Wang, D. Towey, Z. Q. Zhou, and T. Y. Chen, “Metamorphic testing for adobe analytics data collection javascript library,” in *Proceedings of the 3rd International Workshop on Metamorphic Testing*. ACM, 2018, pp. 34–37.

[86] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, “Metamorphic testing: A review of challenges and opportunities,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 4, 2018.

[87] W. E. Howden, “Theoretical and empirical studies of program testing,” in *Proceedings of the 3rd international conference on Software engineering*. IEEE Press, 1978, pp. 305–311.

[88] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[89] F. Pastore, L. Mariani, and G. Fraser, “Crowdoracles: Can the crowd solve the oracle problem,” in *International Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 342–351.

[90] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, “Nezha: Efficient domain-independent differential testing,” in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 615–632.

[91] W. H. Burkhardt, "Generating test programs from syntax," *Computing*, vol. 2, no. 1, pp. 53–73, 1967.

[92] A. G. Duncan and J. S. Hutchison, "Using attributed grammars to test designs and implementations," in *Proceedings of the 5th international conference on Software engineering.* IEEE Press, 1981, pp. 170–178.

[93] D. C. Ince, "The automatic generation of test data," *The Computer Journal*, vol. 30, no. 1, pp. 63–69, 1987.

[94] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, Jul 1990, copyright - Copyright IEEE Computer Society Jul/Aug 1990; Last updated - 2014-05-22; CODEN - IESOEG.

[95] E. Salecker and S. Glesner, "Combinatorial interaction testing for test selection in grammar-based testing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 2012, pp. 610–619.

[96] F. M. Kifetew, R. Tiella, and P. Tonella, "Combining stochastic grammars and genetic programming for coverage testing at the system level," in *International Symposium on Search Based Software Engineering.* Springer, 2014, pp. 138–152.

[97] ——, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empirical Software Engineering*, vol. 22, no. 2, pp. 928–961, 2017.

[98] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, vol. 43, no. 6. ACM, 2008, pp. 206–215.

[99] T. Guo, P. Zhang, X. Wang, and Q. Wei, "Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation," in *Informatics and Applications (ICIA), 2013 Second International Conference on.* IEEE, 2013, pp. 212–215.

[100] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[101] C. Lemieux and K. Sen, "Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 2018, pp. 475–485.

[102] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2018, pp. 2123–2138.

[103] T. Ormandy. (2018, Sep.) halfempty. [Online]. Available: https://github.com/googleprojectzero/halfempty

[104] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *Proceedings of the 23rd international conference on Software engineering.* IEEE Computer Society, 2001, pp. 25–34.

[105] C. Eaton and A. M. Memon, "An empirical approach to evaluating web application compliance across diverse client platform configurations," *International Journal of Web Engineering and Technology*, vol. 3, no. 3, pp. 227–253, 2007.

[106] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *2010 IEEE International Conference on Software Maintenance.* IEEE, 2010, pp. 1–10.

[107] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 561–570.

[108] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-pert: accurate identification of cross-browser issues in web applications," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 702–711.

[109] N. Semenenko, M. Dumas, and T. Saar, "Browserbite: Accurate cross-browser testing via machine learning over image features," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 528–531.

[110] P. Lu, W. Fan, J. Sun, H. Tanaka, and S. Naoi, "Webpage cross-browser test from image level," in *Multimedia and Expo (ICME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 349–354.

[111] J. R. Calamé and J. van de Pol, "Applying model-based testing to html rendering engines–a case study," in *Testing of Software and Communicating Systems*. Springer, 2008, pp. 250–265.

[112] "Mozilla public license version 2.0," Mozilla Foundation. [Online]. Available: https://www.mozilla.org/en-US/MPL/2.0/

[113] "Eclipse public license," Eclipse Foundation. [Online]. Available: https://www.eclipse.org/legal/epl-v10.html