Fuzz Testing
and Test Case Reduction

Ph.D. Dissertation

by

Renáta Hodován
Supervisor: Dr. Tibor Gyimóthy

Submitted to the
Ph.D. School of Computer Science

Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged, 2019
To my Family
When looking back on our lives, it is often hard to determine the point when our current path or profession was chosen. Some people say that they have always known what career they wanted to pursue, others made the decision right before the college application deadline, and there are those who are still uncertain whether they made the right choice even after getting a degree. As for me, the only thing I know for sure is that I have always been fascinated by movies about hackers, even by the not so good ones. I never truly believed that one day I was going to do something similar, but now, 20 years later, here I am. Hunting for issues, reporting security bugs, and writing my thesis. Over the years I slowly realized that nothing could be further from computer science than those movies. However, while sitting in front of my computer, I still often find myself imagine being the protagonist of such a story. And I start to believe that this is reality now.

Getting here has been a long journey, but by no means a lonely one. Many people helped me along the way and I will always be grateful to them. Chronologically first, I would like to say thank you to my high school teachers, Zsuzsanna Bali and Imre Szőke, who persuaded me that girls could also become computer scientists. I am thankful to Tibor Gyimóthy who invited me to join the Department of Software Engineering many years ago and later became my Ph.D. supervisor. I am grateful to him and to Ákos Kiss for their continuous support, especially after I left the safe harbour by choosing a topic that has not had a long research history at our department. I am also thankful to Gábor Lóki and Zoltán Herczeg for putting my research into practice. Last but not least, I would like to thank my family that they provided a pleasant background to my studies throughout the years and also encouraged me to go on with my research.

Renáta Hodován, 2019
## Contents

**Foreword**  

1 Introduction  

I Fuzz Testing of JavaScript Engines  

2 Introduction to JavaScript Fuzzing  

3 Background  

3.1 Fuzz Testing  

3.2 Related Work  

4 Prototype Graph-based Fuzz Testing of JavaScript Engines  

4.1 Type System of the JavaScript Language  

4.2 Graph Representation of Type Information in JavaScript  

4.3 Valid API Call Expressions from a Graph Representation  

4.4 Building a Prototype Graph  

4.5 Evaluation  

4.6 Conclusions  

II Automatic Test Case Reduction  

5 Introduction to Automatic Test Case Reduction  

6 Background  

6.1 Delta Debugging  

6.2 Hierarchical Delta Debugging  

6.3 Related Work  

7 Practical Improvements to DDMIN  

7.1 Parallelization  

7.2 Combination of the Reduce Cases  

7.3 De-prioritization or Omission of “Reduce to Subset”  

7.4 Evaluation  

7.5 Conclusions
8 HDD on Extended Context-Free Grammar-based Parse Trees 45
8.1 Motivation ................................................. 45
8.2 The Modernized HDD ................................... 46
8.3 Evaluation ................................................. 47
8.4 Conclusions ............................................... 52

9 Tree Transformations and Caching for Efficient Reduction 55
9.1 Motivating Example ...................................... 55
9.2 Vertical Tree Squeezing .................................. 56
9.3 Hiding Unremovable Tokens .............................. 57
9.4 Content Caching .......................................... 58
9.5 Evaluation .................................................. 59
9.6 Conclusions .............................................. 62

10 Coarse Hierarchical Delta Debugging 63
10.1 Motivating Example ..................................... 63
10.2 Coarse Hierarchical Delta Debugging ................. 64
10.3 Tree Flattening for Improved Coarse HDD ............ 65
10.4 Evaluation ................................................. 69
10.5 Conclusions .............................................. 73

III Practical Applications 75

11 Fuzzinator 77
11.1 Motivation ................................................. 77
11.2 Architecture ............................................. 78
11.3 Components ............................................. 79
11.4 Example .................................................. 80
11.5 Practical Results ......................................... 84

Appendix 85

A Summary 87
A.1 Summary in English ....................................... 87
A.2 Main Results of the Thesis and Contributions of the Author .................................. 89
A.3 Magyar nyelvű összefoglaló ................................ 91
A.4 Tézispontok és a szerző hozzájárulása az eredményekhez .......................... 94

Bibliography 97
List of Tables

4.1 Size metrics of prototype graphs built for jsc. ........................ 18  
4.2 Code coverage results on jsc after 50,000 generated expressions. 19  
4.3 Number of failures caused in jsc. .................................... 20  
7.1 Test cases, sizes, and their interestingness condition. ............. 36  
7.2 Size of 1-minimal test cases in lines. ................................. 36  
7.3 The number of iterations and test evaluations needed to reach 1-minimality. 41  
7.4 Algorithm variants giving the highest reduction in running time of minimization. .................................................. 43  
8.1 Java example .................................................................. 50  
8.2 HTML example ............................................................... 51  
8.3 HTML & CSS & JS example ............................................... 52  
9.1 Baseline ....................................................................... 60  
9.2 Effect of Squeeze Tree ....................................................... 60  
9.3 Effect of Hide Unremovable Tokens ................................... 60  
9.4 Effect of Content-based Cache .......................................... 60  
9.5 Overall improvement ........................................................ 61  
10.1 Baseline (HDD) ............................................................. 69  
10.2 Coarse HDD ................................................................. 70  
10.3 HDD with Flatten Tree Recursion ..................................... 70  
10.4 Coarse HDD with Flatten Tree Recursion .......................... 70  
10.5 Flatten Tree Recursion + Coarse HDD* + HDD ............... 71  
11.1 Built-in building blocks of Fuzzinator. ............................. 81  
11.2 Issues found in various projects. ........................................ 84
List of Figures

4.1 Example prototype graph manually constructed based on a portion of the ECMAScript 5.1 standard. 13
4.2 Architecture overview of the prototype implementation of the graph-based JavaScript engine API fuzzing technique. 17
4.3 Prototype graph built for jsc with the signature learning technique. 18
7.1 Reduction of the Chromium B test case over iterations. 37
7.2 Size of 1-minimal browser test cases. 38
7.3 Relative size of 1-minimal browser test cases. 39
7.4 Minimization of the WebKit A test case with variations of the sequential algorithm. 40
7.5 Running time of the minimization of the WebKit A test case. 42
8.1 Architecture overview of HDD. 46
8.2 Architecture overview of Picireny. 47
9.1 An HDD example. 56
9.2 The effect of the Squeeze Tree algorithm. 57
9.3 The effect of the Hide Unremovable Tokens algorithm: the tree of Figure 9.1e after the algorithm. 58
9.4 Various configurations tested by HDD during the minimization of the example of Figure 9.1. 59
9.5 Relative number of test cases executed by HDD*. 61
10.1 An HDD example of a JavaScript program that crashes JerryScript. 64
10.2 Reducing the failure-inducing C program with HDD* and Coarse HDD*. 67
10.3 The effect of Tree Recursion Flattening on HDD* and Coarse HDD*. 68
10.4 Results of the reduction algorithm variants relative to HDD*. 71
10.5 The change of test case size over test steps. 72
11.1 The architecture of the Fuzzinator random testing framework. 79
11.2 Screenshot of Fuzzinator. 83
Listings

6.1 The Hierarchical Delta Debugging Algorithm ........................................29
7.1 Non-recursive sequential pseudo-code of $ddmin_2$ .................................32
7.2 Parallel pseudo-code of $ddmin_2$ ..........................................................33
7.3 Parallel pseudo-code of $ddmin_2$ with combined reduce cases ..................34
8.1 Java test case reduced by HDD ...............................................................48
8.2 Java test case reduced by Picireny .........................................................48
8.3 Bison parser rule for Java block .............................................................49
8.4 ANTLR parser rule for Java block ...........................................................49
8.5 Minimized multi-language test case .........................................................52
9.1 The Squeeze Tree Algorithm .................................................................57
9.2 The Hide Unremovable Tokens Algorithm ..............................................58
10.1 The Coarse HDD Algorithm .................................................................65
10.2 C program that crashes GCC 2.95.2. ....................................................66
10.3 The Flatten Tree Recursion Algorithm ..................................................68
11.1 Minimum configuration for fuzzing JerryScript. ......................................81
11.2 Filtering issues. .......................................................................................82
11.3 Extending issues. .....................................................................................82
11.4 Updating the SUT and reducing tests. ....................................................83
“Testers don’t like to break things; they like to dispel the illusion that things work.”
— Kaner, Bach, Pettichord

Introduction

As technology continues to evolve and change, software systems encompass almost every aspect of our life. They are present in our homes, they are used at work, they entertain kids, they are used by doctors for operations, they manage production lines, they predict earthquakes, and all this is just the tip of the iceberg. While these systems make our lives easier, they also act as a potential threat, since they possess or access sensitive information about us. Therefore, security testing has an outstanding importance both to users and vendors of software systems.

When it comes to security testing, the concept of random test generation or fuzz testing [62] turns up quickly, too. Fuzz testing is a popular and automatic negative testing approach, which is based on the idea of generating large numbers of random or semi-random test cases and feeding them to the software under testing (or SUT for short), hoping to trigger some unexpected behaviour. Fuzzing can be used as a standalone testing method but it can also serve as an exploratory step looking for potential entry points of an actual exploit. Its popularity is in great part due to its effectiveness in finding real and reproducible bugs while using fully automated frameworks without the need of professional attendance.

A fuzzer framework usually consists of three main parts: a test generator providing the test cases, a SUT executor functionality which executes and monitors the tested software, and an optional test case reducer, which finds a minimal subset of the issue triggering test case to narrow down the reason of the failure. In this thesis, I focus on the test generator and the test case reducer components of such a framework.

As my primary fuzzing target, I choose JavaScript engines. JavaScript is the de facto standard programming language of web browsers, which became the most popular programming language of the world in the last decade. According to the GitHub statistics from 2018:

“For the sixth year in a row, JavaScript is the most commonly used programming language.” [21]

Looking at the latest survey of StackOverflow from 2018 we can see a quite similar trend:
“JavaScript still has the most contributors in public and private repositories, organizations of all sizes, and every region of the world.” [85]

Because of the prevalence of the language, ensuring the correctness of its execution engines – both functionally and security-wise – is of paramount importance. In this thesis, I introduce a novel fuzzing approach that targets JavaScript engines and has achieved significantly better code coverage than a state-of-the-art JavaScript fuzzing solution, and a prototype implementation that has found more than 100 unique issues in various JavaScript engines to date.

Having an effective random test generator helps us in finding bugs and issue triggering reproduction cases. However, usually only a small portion of these findings are required to reproduce the failure. Throwing away the unnecessary parts helps better understand the problem – which is useful, e.g., for fixing it as soon as possible. Finding that small portion can be really challenging though, even in real life scenarios, not to mention the labor of locating it in a fuzzer generated test case. To lower the amount of work needed for test case minimization, multiple automated solutions have evolved in the last decades.

In this thesis, I investigate the two most widespread reduction algorithms, pinpoint their weaknesses, and propose improvements to them. These improvements got implemented into two open-source tools that were used to evaluate our proposals in practice. The results show that our solutions produce smaller reproduction cases in much less time compared to the baseline approaches.

According to the above introduced topics, the structure of this thesis is split into two main parts and an additional third one containing practical results. In the first part, the focus is on the automated stress testing of JavaScript engine type API. This part introduces a new representation model of JavaScript’s type system which can serve as a test case generation model, too. It will be shown how this model can be automatically extracted from arbitrary engines and how it can be used for test case generation. We will also see that this automatic approach can surpass another state-of-the-art approach.

The second part deals with the subject of automatic test case reduction. I evaluate the popular Delta Debugging and Hierarchical Delta Debugging algorithms and propose multiple improvements to them. The proposed improvements will be justified by extensive evaluations.

The additional third part focuses on the practical aspects of my research. I will introduce Fuzzinator, an open-source fuzzer framework that was used during the experimental evaluations of this dissertation and which assisted me in finding and reporting more than 1000 issues in various projects.

The main parts are followed by an appendix which summarizes the thesis (both in English and in Hungarian).

Finally, the author remarks that although the results presented in this thesis are her major contribution, from this point on, the term ‘we’ will be used instead of ‘I’ for self reference to acknowledge the contribution of the co-authors of the papers that this thesis is based on.
Part I

Fuzz Testing of JavaScript Engines
JavaScript (standardized as ECMA-Script [16] but rarely referred to by that name) is the de facto standard programming language of web browsers, which have evolved into one of the most important application platforms of our days – in which evolution the language played a major role. However, it is not only the client side of the web where JavaScript spreads: it is gaining popularity in server-side scripting as well, thanks to the Node.js framework [72]. Recently, JavaScript has penetrated the embedded world as well: several engines (e.g., Duktape [92], JerryScript [54]) and frameworks (e.g., IoT.js [82]) emerged that enabled the programming of highly resource-constrained Internet-of-Things devices in JavaScript.

Thanks to its manifold application area, it has also been the most popular programming language for years according to GitHub statistics [100]. This popularity comes as no surprise. It is enough to consider how much it facilitates the fast prototyping of ideas via dynamic typing. Its runtime provides fast iteration on those ideas via dynamic compilation. This fuels a fast edit-refresh cycle, which promises an immersive coding experience that is quite appealing to developers.

Because of the prevalence of the language, ensuring the correctness of its execution engines – both functionally and security-wise – is of paramount importance. However, the aforementioned dynamism comes at a price when the API, that is exposed by the underlying JavaScript engine, needs to be tested. All the more so, because the different engines are in different implementation stages of different versions of the standard and because of the different application domains (e.g., web browsers, server side, embedded), the APIs exposed by existing engines differ (and will change as they evolve).

When it comes to security testing, the concept of random test generation or fuzz testing turns up quickly, too. Fuzz testing is a popular automatic negative testing approach based on the idea of generating a large number of random or semi-random test cases and feeding them to the software under testing (or SUT for short), hoping to trigger some unexpected behavior. Fuzzing can be used as a standalone testing method but it can also serve as an exploratory step looking for potential entry points of an actual exploit. The test generation is either generation or mutation-based, meaning
that the test cases can be generated from scratch, according to a model, or by mutating existing test cases by applying transformations to them. Depending on the precision of the model or the transformations, the generated test cases will be able to exercise the SUT at different abstraction levels. For example, a completely random model that has no information about the expected input format at all and generates random sequences only, will not be able to test deeper code paths than the parser. On the other hand, a too restrictive model does not leave much room for the generator to capitalize on randomization power.

In this part, a generation-based fuzz testing approach will be introduced to exercise JavaScript engine APIs, which automatically builds its generator model and aims at keeping the balance between randomization and overfitting. We formally define a graph representation that is suited to describe type information in an engine, explain how to build such graphs, and describe how to use them for API fuzz testing. Our experimental evaluation of the technique on a real-life in-use JavaScript engine shows that the introduced approach gives better coverage than the available state-of-the-art fuzzer and could also find valid issues in the tested system.

The rest of this part is organized as follows: Chapter 3 gives an overview about fuzz testing in general and introduces the basics of the type system of the JavaScript language, while Chapter 4 introduces the idea of Prototype Graph-based fuzz testing of JavaScript engines and evaluates it on real-life applications.
“No amount of testing can prove a software right, a single test can prove a software wrong.”

— Amir Ghahrai

3 Background

3.1 Fuzz Testing

Fuzzing [89], or random testing, is a popular technique as it promises the creation of a large number of test cases with limited effort. Moreover, as a result of randomness, it is often capable of generating extreme test cases that are easily overlooked by a human test engineer. Of course, there is a limit to the application area of such ‘blindly’ generated test cases as it is difficult to decide in general whether the program that consumed the random input executed it in a semantically correct way. However, some important aspects of the execution can still be monitored, such as whether the program accessed resources (e.g., code, stack, heap memory) according to its permissions or adhered to its design contracts (e.g., assertions). This makes the technique highly useful for security testing.

The first fuzzers were truly random, simply feeding random byte sequences to their system under testing. Although such simple fuzzers can reveal some surprising faults, they most often cannot scratch anything but the surface of SUTs that have a complex input structure. The test cases generated by such format-unaware fuzzers are usually caught and discarded by the input parser, so deeper code paths do not get exercised. Model-based fuzzers aim at reaching these deeper parts of the SUT by generating test cases from some model of the input format, trying to ensure that the high-level structures that get checked by the SUT’s parser are valid while the ‘real’ content gets randomized. Model-based fuzzers are usually sorted into two main categories based on how they build their model and generate test cases from it: purely generation-based fuzzers expect an explicit representation of the input format, usually written by hand, while purely mutation-based approaches need a suite of existing test cases, which act as an implicit representation of the input format and can be mutated or recombined to yield new inputs. Both approaches have their advantages and disadvantages. In case of complex input formats, building a model for a generation-based fuzzer is a tedious and time-consuming task: e.g., generating random programs as test inputs (like JavaScript sources) for compilers or interpreters (e.g., for JavaScript engines) needs the modelling of syntax requirements, symbol tables, type information, etc. On the other hand,
mutation-based fuzzers require a vast number of existing test cases to work with (and they may still need additional information on where and how to mutate or recombine them).

In this work, a generation-based approach will be presented, where the model is automatically inferred either from the target application only or with additional information extracted from existing test cases.

3.2 Related Work

The origins of random test generation date back to at least the '70s, when Purdom published his paper about grammar-based sentence generation for parser testing [77]. However, the term fuzzing was coined by Miller in 1988 only, as a result of truly random noise appearing in modem lines, disturbing terminals, and causing programs to crash. Since then, randomized testing has become widespread; a good overview is also given in the work of Sutton et al [88].

Fuzzing techniques are usually sorted into the blackbox, graybox and whitebox categories based on the amount of information they possess about the software under testing. Blackbox approaches do not know any implementation details about the target, they may only have knowledge about the expected input format. Graybox fuzzers do not have access to the source code either, but they have some kind of instrumentation on the target that can feedback information, usually coverage, about the execution. Lastly, whitebox fuzzers have access to the source of the tested application and can use this information to guide the generation.

Besides the above three categories, fuzzers can also be distinguished based on the way they create new test cases. Mutational approaches require having existing test cases that will be mutated during a fuzz session. Generation-based approaches, on the other hand, work with models describing the expected format of the input.

Zzuf [37] is a simple mutational blackbox fuzzer which creates new tests by applying random bit flips to existing test cases. Peach [75] is one of the most popular model-based black-box fuzzer with both generational and mutational abilities. It works with XML-based PIT files describing both the expected input format and the way the target should cooperate with the fuzzer.

Holler et al. created LangFuzz [50], a language-independent approach that parses an initial test suite with ANTLR grammars and extracts code fragments from it. These fragments are then reused and recombined to create new inputs. Although the idea is language independent, the primary use of the prototype tool is to test SpiderMonkey, the JavaScript engine of FireFox. IFuzzer [93] is an evolutionary fuzzer built upon the idea of LangFuzz. It defines a fitness function using information about the generated test (like complexity metrics) and the feedback of the SUT (like crashes, exceptions, timeouts, etc.) to choose the elements of the next population. BlendFuzz [97] is also similar to LangFuzz as it processes existing sources with ANTLR grammars, but it also improves the mutation phase by collecting additional information from parse trees. CSmith [98] is a C compiler fuzzer which can generate C99-conform sources without any undefined behavior.

Skyfire [94] is used to generate initial seeds for other fuzzing approaches. It learns probabilistic context-sensitive grammars from samples that are used later to generate the seeds. Skyfire was used to test the JavaScript engine of Internet Explorer 11. Similarly, SmartSeed [61] aims to generate high-value binary seeds using a machine learning
model. Höschelle and Zeller [51] created a tool named AutoGram that automatically inferred context-free grammars of inputs of Java programs using dynamic taint analysis. This inferred grammar could be used later as a generation model in a fuzzing session.

Closest to our work in Chapter 4 is the jsfunfuzz [70] system from Ruderman, which we have chosen to compare our proposed technique against. It is a manually crafted JavaScript fuzzer used to test SpiderMonkey. This system applies an engine discovery approach similar to ours, but it does not utilize all introspecting possibilities of the language, e.g., it does not traverse the prototype chains and it completely omits the automated discovery and learning of function signatures (all possible parametrizations are manually specified for the fuzzer).

Recently, the most popular and oft-cited works are the mutation-based graybox approaches. One of the most famous among them is American Fuzzy Lop [99] (or AFL for short). AFL uses lightweight instrumentation to access code coverage information that is used to decide which mutant is worth to be kept for further transformations. AFL has several variants. AFLFast [6] uses a Markov-chain-based power schedule on path frequency to compute the times of random mutations for each seed. AFLGo [5] proposes simulated annealed-based power schedule to guide mutation. FairFuzz [57] changes the mutation by only fuzzing seeds hitting rare branches. Driller [86] augments AFL with symbolic executions. EnFuzz [12] aims to integrate multiple fuzzing strategies to obtain better performance. Rajpal et. al. [79] use neural networks to learn patterns in the input files from past fuzzing explorations to guide future fuzzing steps. VUzzer [80] is also a feedback-driven fuzzer, which uses data and control flow information to decide where and what to mutate and to prioritize deep and deprioritize frequent paths.

Honggfuzz [27] and libFuzzer [60] are also coverage-guided graybox fuzzers but they run in-process and they speed up fuzzing by focusing only on the target functions. Google’s OSS-Fuzz [28] platform uses both AFL and libFuzzer to test open-source projects with a large user base.

The fuzzers in the third category use whitebox approaches, mostly based on symbolic execution and taint analysis. In general, these approaches use code instrumentation or program tracing to obtain information of the program execution to guide test case generation. This technique has been implemented in numerous tools, such as CUTE [84], DART [23], SAGE [24, 22, 25], SmartFuzz [67], EXE [28], KLEE [10], BitFuzz [9], and TaintScope [96].
“The strength of JavaScript is that you can do anything. The weakness is that you will.”
— Reg Braihwaite

4

Prototype Graph-based Fuzz Testing of JavaScript Engines

4.1 Type System of the JavaScript Language

The type system of JavaScript is often being referred to as *untyped, dynamic, weakly typed* or *object-based*. In this context, *untyped* means, that JavaScript has no *static types*, instead everything belongs to a single type. This does not mean that JavaScript values are not tagged with types, but these tags are set dynamically at runtime. This is where the *dynamically typed* or *dynamically checked* properties come from. The *weakly typed* property means, that JavaScript permits a large number of implicit type conversions, which enables making operations with non-matching types by automatically doing type-coercion. The *object-based* property refers to the fact that JavaScript does not possess the traditional class-based type idea, but instead it uses so-called *prototype-chains* (and *members*) to sort objects into similarity categories or ‘types’. Note that *object-based* is not equivalent to *object-oriented*, since inheritance is not realized through the traditional *class-based* approach but it uses a *prototype-based* one. It still stands for the ECMAScript 2015 [18] standard which introduces the *class* keyword but it is just a syntactic sugar over the original *prototype-based* solution. *Prototype-chains* are the way of realizing inheritance in JavaScript. Each object in JavaScript has a dynamic bag of *own properties* holding links to other objects and a private `__proto__` field referring to the object’s *prototype*. The prototype object also can have its own prototype object, and so on until *null*, the final link in this *prototype-chain*, is reached. Roughly, all objects are descendants of the *Object.prototype*, which is the top of the *prototype-chain*. When a property access is tried on an object, then not only the object’s *own properties* are looked up but those of its prototype’s and those of its prototype’s prototype, and so on, until a matching property is found or the end of the chain is reached.

Functions in JavaScript behave like other objects. They can be inherited or extended, they can have *own properties*, and they have a prototype. If a function is called as a constructor, i.e., the call is prefixed with the *new* keyword, then the output object’s prototype (the object pointed to by its `__proto__` field) will refer to the
object linked by the prototype field of the called functions. (These two fields must not be confused, **__proto__** always points to the inherited properties while prototype describes the inheritable properties in case of constructor calls.)

In the next section, a graph representation will be introduced to model this structure in a way that is suitable for random JavaScript call generation, and hence, can stress-test an implementing engine.

### 4.2 Graph Representation of Type Information in JavaScript

In this section we define a graph, titled the Prototype Graph, that can represent the type system described in Section 4.1.

**Definition 4.1 (Prototype Graph)** Let a Prototype Graph be a labeled directed multigraph (a graph allowing parallel edges with own identity)

$$G = \langle V, E, s, t, l_{\text{prop}}, l_{\text{param}} \rangle$$

such that

- $V = V_{\text{type}} \cup V_{\text{sig}}$, set of vertices, where the subsets are disjoint,
  - $V_{\text{type}}$ vertices represent ‘types’, i.e., categories of similar objects,
  - $V_{\text{sig}}$ vertices represent ‘signatures’ of callable types, i.e., functions,
- $E = E_{\text{proto}} \cup E_{\text{prop}} \cup E_{\text{cstr}} \cup E_{\text{call}} \cup E_{\text{param}} \cup E_{\text{ret}}$, set of edges, where all subsets are mutually disjoint,
  - $E_{\text{proto}}$ edges represent prototype relation (‘inheritance’) between types,
  - $E_{\text{prop}}$ edges represent the properties (‘members’) of types,
  - $E_{\text{cstr}}$ and $E_{\text{call}}$ edges connect callable types to their signatures and represent the two ways they can be invoked, i.e., the construct and call semantics,
  - $E_{\text{param}}$ edges represent type information on parameters of callable types,
  - $E_{\text{ret}}$ edges represent return types of callable types,
- $s : E \rightarrow V$ assigns to each edge its source vertex, under the constraint that $\forall e \in E_{\text{proto}} \cup E_{\text{prop}} \cup E_{\text{cstr}} \cup E_{\text{call}} \cup E_{\text{param}} : s(e) \in V_{\text{type}}$ and $\forall e \in E_{\text{ret}} : s(e) \in V_{\text{sig}}$,
- $t : E \rightarrow V$ assigns to each edge its target vertex, under the constraint that $\forall e \in E_{\text{proto}} \cup E_{\text{prop}} \cup E_{\text{ret}} : t(e) \in V_{\text{type}}$ and $\forall e \in E_{\text{cstr}} \cup E_{\text{call}} \cup E_{\text{param}} : t(e) \in V_{\text{sig}}$,
- the $\langle V, E_{\text{proto}}, s|_{E_{\text{proto}}}, t|_{E_{\text{proto}}} \rangle$ directed sub-multigraph is acyclic,
- $l_{\text{prop}} : E_{\text{prop}} \rightarrow \Sigma$ labeling function assigns arbitrary symbols (‘names’) to property edges, under the constraint that $\forall e_1, e_2 \in E_{\text{prop}} : s(e_1) = s(e_2) \Rightarrow l_{\text{prop}}(e_1) = l_{\text{prop}}(e_2) \iff e_1 = e_2$,
- $l_{\text{param}} : E_{\text{param}} \rightarrow \mathbb{N}_0$ labeling function assigns numeric indices to parameter edges, under the constraint that $\forall e_1, e_2 \in E_{\text{param}} : t(e_1) = t(e_2) \Rightarrow l_{\text{param}}(e_1) = l_{\text{param}}(e_2) \iff e_1 = e_2$. 

12
Informally, a prototype graph is a collection of type and sig vertices connected by six different kinds of edges (and multiple edges can run between two vertices). Proto and prop edges connect type vertices, while the others connect type and sig vertices in one direction or the other. And finally, member name information and function argument order is encoded in edge labels. (Note that vertices have no labeling; most information is stored in the existence of and labels of edges.)

As an example, Figure 4.1 shows a prototype graph of 6 type and 2 sig vertices, manually constructed based on a portion of the ECMAScript 5.1 standard. The graph contains the types of Object, Object.prototype, Function, and Function.prototype objects, the global object, and also the constructor signature for Object and the call signatures for three functions of it.

Finally, we define some useful notations on prototype graphs as follows.

**Definition 4.2 (Notations)** Let \( G = \langle V, E, s, t, l_{\text{prop}}, l_{\text{param}} \rangle \) be a prototype graph, where \( V = V_{\text{type}} \cup V_{\text{sig}} \) and \( E = E_{\text{proto}} \cup E_{\text{prop}} \cup E_{\text{cstr}} \cup E_{\text{call}} \cup E_{\text{param}} \cup E_{\text{ret}} \), according to Definition 4.1. Then we introduce the following notations:

- Let \( v \xrightarrow{x} v' \), denote the edge \( e \) of type \( x \), where \( x \in \{\text{proto, prop, cstr, call, param, ret}\} \), iff \( e \in E_x \land s(e) = v \land t(e) = v' \).

- Let \( v \xrightarrow{e_1 \ldots e_n}_X \ast v' \), denote the finite path \( e_1 \ldots e_n \) over \( X \) type of edges, where
4.3 Valid API Call Expressions from a Graph Representation

In this section, we present an application of the prototype graph, i.e., JavaScript engine API fuzzing, as motivated in the introduction. Thus, given a built graph, our goal is to generate call expressions that invoke functions on API objects with type-correct arguments. Therefore, we formally define with graph terms how such expressions can look like.

**Definition 4.3 (Function Call Expressions)** Let $G = \langle V, E, s, t, l_{prop}, l_{param} \rangle$ be a prototype graph, where $V = V_{type} \cup V_{sig}$, $E = E_{proto} \cup E_{prop} \cup E_{cstr} \cup E_{call} \cup E_{param} \cup E_{ret}$, and $l_{prop} : E_{prop} \to \Sigma$, according to Definition 4.1. Let $\Lambda : \mathcal{P}(V) \to \mathcal{P}(\Sigma^{+})$ be a function that maps a set of types to a set of literals from an extended alphabet $\Sigma^{+} \supseteq \Sigma \cup \{\text{`new'}, \text{`.'}, \text{`(.'), (., .), (., .).}'\}$. Then we define $\Phi_{G,A} : V_{type} \to \mathcal{P}(\Sigma^{+})$ as a function that gives for some selected starting vertex $v_0$ a set of type-correct call expressions that are available from an object of that type:

$$\Phi_{G,A}(v_0) = \Gamma_{G,A}(v_0, V_{sig})$$

$\Phi_{G,A}$ is given with the help of several auxiliary functions, all of which are defined below:

- The function $\Gamma_{G,A} : V \times \mathcal{P}(V) \to \mathcal{P}(\Sigma^{+})$ gives the set of type-correct expressions available along a set of paths:

$$\Gamma_{G,A}(v_0, V') = \bigcup_{e_1 \ldots e_n \in \Pi_{G}(v_0, V')} \left( (\text{`new'}, \ldots, \text{`(.'}) \cdot \Lambda((\{v_0\}) \cdot \prod_{i=1}^{n} \text{`.'}) \cdot \gamma_{G,A}(v_0, e_i) \right)$$

- The function $\gamma_{G,A} : V \times E \to \mathcal{P}(\Sigma^{+})$ gives a set of sub-expressions for a step (i.e., edge) along a path:

$$\gamma_{G,A}(v_0, e) = \begin{cases} 
\text{`.'} \cdot l_{prop}(e) & \text{if } e \in E_{prop}, \\
\text{`(.'} \cdot \left( \prod_{i=1}^{n} \text{`.'} \cdot \Gamma_{G,A}(v_0, \nabla_{G}(v_i)) \cup \Lambda(\nabla_{G}(v_i)) \right) \cdot \text{`.}') & \text{if } e \in E_{cstr} \cup E_{call}, \text{ where } \\
\lambda & \text{if } \forall 1 \leq i \leq n : v_i \xrightarrow{\text{`param} e_i}{e_i / \text{`param} \text{.}}} t(e) & \text{and} \\
\lambda & \text{if } \forall 1 \leq i < n : l_{param}(e_i) < l_{param}(e_{i+1}), \\
\lambda & \text{otherwise.}
\end{cases}$$

- The function $\Pi_{G} : V \times \mathcal{P}(V) \to \mathcal{P}(P_{G})$ gives the set of all finite paths from a vertex to a set of vertices over proto, prop, cstr, call, and ret edges:

$$\Pi_{G}(v, V') = \left\{ e_1 \ldots e_n : v \xrightarrow{\text{proto,prop,cstr,call,ret}} v' \land v' \in V' \right\}.$$
Chapter 4. Prototype Graph-based Fuzz Testing of JavaScript Engines

- The function $\nabla_G : V \to \mathcal{P}(V)$ gives the set of all vertices reachable from a given vertex backwards over proto edges:

$$\nabla_G(v) = \left\{ v' : \exists e_1 \ldots e_n : v' \xrightarrow{\text{proto}} v \right\}.$$

- The $\cdot$, $\prod$, and power notations denote concatenation of strings (or sets of strings) over an alphabet, and $\lambda$ is the empty word. The $\prod$ notation with an additional superscript symbol denotes concatenation with a separator symbol:

$$\prod_{i=1}^{n} a_i = \begin{cases} \left( \prod_{i=1}^{n-1} a_i \right) \cdot \sigma \cdot a_n & \text{if } n > 1, \\ \prod_{i=1}^{n} a_i & \text{otherwise.} \end{cases}$$

- $[P]$ denotes the Iverson bracket, i.e., gives 1 if $P$ is true, 0 otherwise.

As visible from the definition above, the graph representation is capable of describing how a property can be accessed (the first case of $\gamma_{G,A}$), how a function can be parametrized to retrieve a type-correct value (the second case of $\gamma_{G,A}$), and how a new object can be created with a constructor call (the $E_{\text{cstr}}$-related parts of $\Gamma_{G,A}$). There is one more way of creating an expression of a given type, however: with literals. Since they fall outside the expressiveness of the graph representation – mostly because a literal is more of a syntactic entity while the concepts in the graph represent components of type information – possible literals of a type (or some types) are represented by the $\Lambda$ function that complements the graph.

An actual implementation of the above formalism, e.g., a fuzzer, would most probably not generate the (potentially infinite) set $\Phi_{G,A}(v_0)$ but may choose an arbitrary element from it (and perhaps a different one on every execution). That can be achieved by a random walk on the prototype graph based on the informal concept behind the formal definition: “First walk forward on proto, prop, cstr, call, and ret edges to a sig vertex, then walk backward on param and proto edges, and so on...”. Also, in an API fuzzer, $v_0$ would most probably be the type of the global object, and $\Lambda$ would map $v_0$ to a literal referring to the global object.

As an example, below we give some expressions in $\Phi_{G_{ex},A_{ex}}(v_{ex})$, where $G_{ex}$ is the graph in Figure 4.1, $v_{ex}$ is the type of the global object in that graph (marked with black), and $A_{ex}(V') = \bigcup_{v \in V'} \{ 'this' \}$:

- `this.Object.getOwnPropertyDescriptor(this.Function.prototype),`
- `new (this.Object)(this).`

4.4 Building a Prototype Graph

The most trivial way of building a prototype graph may seem to be by hand. However, processing web standards manually – not only that of ECMAScript, but potentially others as well, e.g., HTML5 DOM bindings – may require huge human efforts, is still heavily error prone, will be hard to maintain, and will not know about any engine-specific extensions. Thus, we outline two automated techniques below.
Both approaches rely on the introspecting capabilities of the JavaScript language: not only can we determine the actual type of every expression at run-time (with the `typeof` operator, e.g., whether it is of primitive boolean, number, or string type, or an object or a function, to name the most important types) but we can enumerate all properties and walk the property chains of all objects (with `Object.getOwnPropertyNames()` and `Object.getPrototypeOf()`), and retrieve the number of the formal parameters of every function (by reading their `length` property). By relying on introspection, the approaches do not require the source code of the engine-to-be-tested making them applicable to any engine.

The first approach, which we will call *engine discovery* in this thesis (or just discovery, for short), is a one-time analysis of the engine. Its core idea is to execute a specially crafted JavaScript program inside the engine-to-be-tested using the above mentioned constructs. If this JavaScript program gets access to an object, it can create type descriptor data structures from all values recursively reachable from it by recording property names, and prototype and property relations, – as well as formal argument list lengths for function objects. If this JavaScript program gets access to the global object of the execution environment then it can discover the whole API of the engine. Fortunately, the language standard mandates the existence of that global object (available as `this` in the outmost lexical scope), thus the approach is universal for all engines.

The discovery technique can find the prototypes and properties in the exposed API, but as mentioned above, it has a very limited view of the signatures (parameter and return types) of the functions. Therefore, we propose a second approach to extend the information about functions in the type descriptor data structure: in JavaScript, every object can be modified dynamically, even most of the built-in objects, and this includes the replacement of built-in functions with user-written JavaScript code. Thus, we propose to wrap (or patch) every function found using the discovery technique in a special code that, when called, collects type descriptor information about every parameter before passing them through to the original function, and also collects details about the returned value after the original call finished but before it is given back to the caller. This way the observable behaviour of the wrapped system does not change, but as programs execute in the engine, information about function signatures in the type descriptors can be continuously extended, refined (or, learned). Executing official or project-specific JavaScript test suites in such a patched environment is a plausible choice to learn signatures.

The data structures built by the engine discovery and signature learning approaches can be exported from the engine, e.g., in JSON format by using built-in conversion routines. Then, that information can be easily transformed to the prototype graph format introduced in Section 4.2.

### 4.5 Evaluation

To evaluate the ideas explained in the previous sections, we have created a prototype implementation. The code that discovers the API of the engine and learns signatures from existing tests was written in JavaScript (relying on the introspection capabilities of the language, as described in Section 4.4), while test generator code, execution harness for the engine-under-test, and utility routines were implemented in Python 3, with the help of the graph-tool module [76]. The architecture overview of the prototype
Figure 4.2. Architecture overview of the prototype implementation of the graph-based JavaScript engine API fuzzing technique. The black boxes stand for the SUT, white elements are part of the implementation, while gray ones are generated during its execution. Dashed lines separate parts which execute only once from those which run multiple times (changing elements are marked with an asterisk).

implementation is shown in Figure 4.2.

As SUT, we have chosen \textit{jsc}, the command line JavaScript execution tool from the WebKit project [3]. The project – checked out from its official code repository at revision 192323 dated 2015-11-11 – was built in debug configuration for its GTK+ port (i.e., external dependencies like UI elements were provided by the GTK+ project [32]), on an x86-64 machine running Ubuntu 14.04 with Linux kernel 3.13.0, with gcc 4.9.2. To enable the investigation of the effects of fuzzing on the SUT, we made use of the coverage computation support of the compiler together with the gcov 4.9.2 and gcovr 3.2 tools [20].

Our goal was not only to evaluate the prototype graph-based fuzzing technique on its own but also to compare it to existing solutions. Therefore, we have downloaded jsfunfuzz [70] – with hash 6952e0f dated 2016-01-08 – one of the few available open source JavaScript fuzzers to act as a baseline for our experiments.

Since in our fuzzing technique the first step is to build a prototype graph, we have applied both previously outlined approaches to \textit{jsc}, i.e., first we have discovered the engine and then gathered functions signatures from existing tests. For the latter step, we have used the official JavaScript stress test suite of the WebKit project, executing 3,573 tests in the wrapped engine.

Table 4.1 shows the size metrics of the built graphs. The most striking difference is in the number of $V_{\text{type}}$ vertices. This can be attributed to two factors: first, because of the limited view of the engine discovery on the function signatures, a large number of function objects are considered to be of the same prototype (i.e., only their argument number differentiates between them, and those with the same number of arguments get represented by a single type vertex). With signature learning, however, most of the
Chapter 4. Prototype Graph-based Fuzz Testing of JavaScript Engines

Table 4.1. Size metrics of prototype graphs built for jsc. (Signatures were learned from 3,573 official stress tests.)

<table>
<thead>
<tr>
<th></th>
<th>pgdiscover</th>
<th>pglearn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V_{type}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>V_{sig}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>E_{proto}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>E_{prop}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>E_{cstr}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>E_{call}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>E_{param}</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>E_{ret}</td>
<td>$</td>
</tr>
</tbody>
</table>

Figure 4.3. Prototype graph built for jsc with the signature learning technique.

function objects get represented by a separate type vertex. The second factor in the increase are new types emerging during signature learning either originating from the test suite (e.g., previously unseen parameters) or from the engine itself (e.g., natively constructed return values). Figure 4.3 shows the prototype graph resulting after the signature learning step.

Once the graphs were built, we have used our test generator implementation to create 50,000 expressions from both, each expression resulting from a path consisting of a maximum of 20 property steps (as defined by $\gamma_{G,A}$ in Section 4.3). Then, we
executed all expressions in *jsc*, one by one, and monitored their results. We took note of all assertion failures and program crashes, and after every 5,000 expressions we also measured the accumulated code coverage of the hitherto executed tests.

For the execution of *jsfunfuzz*, we used its own framework with the slight modification of adding a periodic code coverage measurement to the system. *Jsfunfuzz* was written with continuous (or continuously restarting) execution in mind and thus does not support the generation of a given number of expressions. Therefore, we could neither generate one expression per test as with our implementation nor could we execute exactly 50,000 expressions. However, we wanted to keep changes to the minimum, therefore, we did not alter that behavior. Since *jsfunfuzz* did not support any command line argument to limit the number of generated expressions, we monitored the output directory and stopped the whole framework after the first code coverage measurement past the 50,000 expressions limit (thus finishing with 50,320 expressions).

Table 4.2 shows the line coverage results of all three fuzzing approaches both with module-level granularity and in total. (We have considered each sub-directory and standalone file of the *JavaScriptCore* build folder of the WebKit project as a separate module.) The results show that the basic engine discovery-based approach does not perform as well as *jsfunfuzz* in terms of total code coverage (23.31% compared to 37.25%), but as soon as we extend our graph with signature information extracted
Chapter 4. Prototype Graph-based Fuzz Testing of JavaScript Engines

Table 4.3. Number of failures caused in \textit{jsc}.

<table>
<thead>
<tr>
<th></th>
<th>pgdiscover</th>
<th>pglearn</th>
<th>jsfunfuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>total failures</td>
<td>1326</td>
<td>1445</td>
<td>4</td>
</tr>
<tr>
<td>unique failures</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

from tests, it improves significantly and gives higher results (44.13%).

We should also highlight results on three important modules, namely on \textit{runtime}, \textit{yarr}, and \textit{jsc.cpp}. The first module contains C++ implementations of core JavaScript language functionality (like built-in functions), while the second contains the regular expression engine of the project. The third module (actually, a single file), is the main command line application, which is a classic JavaScript engine embedder in the sense that it binds some extra, non-standard routines into the JavaScript environment. That is, these are the modules that expose API of native code to the JavaScript space and thus are in our focus. As the table shows, even the simpler engine discovery-based technique can outperform jsfunfuzz in two out of these three modules, and the signature-extended variant gives the best results in all three cases.

For the sake of completeness, we should also mention some modules where the coverage is very low for all techniques. \textit{API} and \textit{bindings} contain API exposed to an embedder: JavaScript code has no effect on how the engine is linked to its container application (moreover, \textit{bindings} is marked deprecated in the code base, thus will not reach a higher coverage ever). The other modules – i.e., \textit{debugger}, \textit{disassembler}, \textit{inspector}, \textit{profiler}, and \textit{tools} – contain code that are development-related and are also not under the control of the JavaScript code. Thus, these modules cannot be – and are not – the target of JavaScript API fuzzing.

As the ultimate goal of fuzzing is not only to reach good code coverage but also to cause system malfunction, we compared the three techniques on the basis of caused crashes as well. Table 4.3 shows the total number of observed failures, and since several tests triggered the same problem, the number of unique failures as well. (The uniqueness was determined based on crash backtrace information retrieved with a debugger.) Interestingly, both graph-based fuzzing techniques found the same failures. This also means that even the engine discovery technique with lower total coverage ratio could find more errors than jsfunfuzz. (However, it has to be noted that one of the two crashes caused by jsfunfuzz was not found by the graph-based prototype implementation.)

4.6 Conclusions

In this chapter, we have defined a graph-based formalization of type information in JavaScript, we have shown how to automate the building of such graphs to describe the API exposed by a JavaScript execution engine, and we have also defined an API fuzzing method based on graph terms. To the best of our knowledge, this is the first work to use graph formalization and traversal for representing type information of JavaScript and use it for fuzzing. In addition to the formal definitions, we have also presented the experimental results of a prototype implementation of a JavaScript engine API fuzzer. The results show that the prototype graph based API fuzzing technique can be on par with available solutions in terms of total code coverage and even yield
significantly better coverage results in JavaScript API-related modules. Moreover, the implementation triggered real program crashes during the experiments (and later, too). The prototype implementation has been actively used ever since the publication of the presented research and it has found more than 100 unique issues in the JerryScript, Duktape, and IoT.js JavaScript engines and platform, which were reported to and fixed by the developers of the named projects.
Part II

Automatic Test Case Reduction
“Inside every large program, there is a small program trying to get out.”

— C. A. R. Hoare

5

Introduction to Automatic Test Case Reduction

It is considered a common knowledge that there is no such thing as a bug-free program. Bugs – and bug-inducing test cases – can be uncovered by users, by developers, or by various automated testing frameworks, just like it was shown in Part I. However, usually only a small portion of these findings are required to reproduce the failure. Throwing the unnecessary parts away helps better understand the problem – which is useful, e.g., for fixing it as soon as possible. Finding that small portion can be really challenging though, even in real life scenarios, not to mention the labor of locating it in a fuzzer-generated test case. The importance of test case reduction is emphasized by many communities expecting bug reports from their users. The developers of Gecko – which is the rendering engine of the Firefox browser – claim that

“Every hour Gecko engineers spend decomposing bug reports is an hour they can’t spend fixing bugs.” [69]

Similarly, the issue reporting guideline of WebKit – the engine under the hood of the Safari browser – says:

“With a reduced test case, the development team will spend less time identifying the problem and more time determining the solution.” [4]

Google Chrome developers also expect simplified test cases in bug reports:

“If a bug can be reduced to a simplified test, then create a simplified test and attach it to the bug.” [90]

Of course, not only browser vendors prefer receiving reduced test cases but many others, too, like GCC:

“Our bug reporting instructions ask for the preprocessed version of the file that triggers the bug. Often this file is very large; there are several reasons for making it as small as possible.” [19]
Similarly, LLVM developers ask to:

“narrow down the bug so that the person who fixes it will be able to find the problem more easily.” [59]

And jQuery guidelines say:

“Reduce, Reduce, Reduce! ... Bugs that are reported without reduced test cases take on average much longer to fix than bugs that are submitted with them.” [91]

As it can be seen, test case reduction is actively present and useful in many application areas. Although the quotes above suggest to reduce file-based inputs, reduction approaches can be utilized to simplify other formats, too, like UI [33] or Android event sequences [53, 13], failure-inducing code changes or program statements, the smallest subset of a network traffic causing issues [78], etc. Even though the application area is manifold, reduction in general can be defined as an iterative application of various transformations. These transformations can be as simple as deletions, but can also be other changes that can decrease the size or complexity metric of the test under reduction.

In practice, test case reduction can happen either manually or automatically. Automated approaches tend to mimic the same behaviour as a human tester would do but without consuming a lot of valuable manpower. Many automated approaches are based on the divide-and-conquer idea: the input is iteratively divided into two parts, one for keeping and another for removal. If the kept part still produces the original behaviour, then it will be reduced further. Otherwise, the removal will be revoked and another partitioning will be tried. In this chapter, two of the most oft-cited automated reduction algorithms, Delta Debugging and Hierarchical Delta Debugging, will be evaluated and revised both output quality and performance-wise.

The rest of this part is organized as follows: first, Chapter 6 will give a brief overview of the two underlying algorithms, Delta Debugging and Hierarchical Delta Debugging, then Chapter 7 discusses our observations on the original \texttt{ddmin} algorithm and presents improved variants. Chapter 8 identifies issues with the original Hierarchical Delta Debugging approach and describes how to deal with them. Chapter 9 will introduce such general tree-based input preprocessing possibilities that can significantly decrease the duration of reduction, and finally, Chapter 10 describes the idea of Coarse HDD.
“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

— Brian W. Kernighan

6 Background

6.1 Delta Debugging

One of the very first and most oft-cited research in the field of automatic test case reduction is the more than a decade old idea of Delta Debugging from Zeller and Hildebrandt [102, 36, 103]. Delta Debugging has two variants, the minimizing Delta Debugging algorithm abbreviated as \textit{ddmin} and the general Delta Debugging version, called \textit{dd}. \textit{ddmin} is used to simplify failure-inducing test cases, while \textit{dd} can be applied to isolate failure-inducing differences if a passing version of the input is available, too. In this research, the focus is on the \textit{ddmin} variant.

Informally, the \textit{ddmin} algorithm takes a set, or configuration, representing a failure-inducing test case as input and iteratively removes subsets of this configuration while keeping the remaining test case failure-inducing. Usually, a configuration is composed of the lines or characters of a text file, and the removal of configuration subsets means actual deletion of parts from the content.

Definition 6.1 (Zeller and Hildebrandt’s)

Let \textit{test} and \( c_X \) be given such that \( \textit{test}(\emptyset) = \checkmark \land \textit{test}(c_X) = \times \) hold. The goal is to find \( c'_X = \text{\textit{ddmin}}(c_X) \) such that \( c'_X \subseteq c_X \), \( \textit{test}(c'_X) = \times \), and \( c'_X \) is 1-minimal. The minimizing Delta Debugging algorithm \textit{ddmin}(c) is

\[
\text{\textit{ddmin}}(c) = \text{\textit{ddmin}}_2(c_X, 2) \quad \text{where}
\]

\[
\text{\textit{ddmin}}_2(c'_X, n) = \begin{cases} 
\text{\textit{ddmin}}_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot \text{\textit{test}}(\Delta_i) = \times \text{ ("reduce to subset")}, \\
\text{\textit{ddmin}}_2(\nabla_i, \max(n - 1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \cdot \text{\textit{test}}(\nabla_i) = \times \text{ ("reduce to complement")}, \\
\text{\textit{ddmin}}_2(c'_X, \min(|c'_X|, 2n)) & \text{if } n < |c'_X| \text{ ("increase granularity")}, \\
\text{\textit{ddmin}}_2(c'_X, \argmin(|\Delta_i|, n)) & \text{otherwise ("done")},
\end{cases}
\]

where \( \nabla_i = c'_X - \Delta_i \), \( c'_X = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n \), all \( \Delta_i \) are pairwise disjoint, and \( \forall \Delta_i \cdot |\Delta_i| \approx |c'_X|/n \) holds. The recursion invariant (and thus precondition) for \textit{ddmin}_2 is \( \text{\textit{test}}(c'_X) = \times \land n \leq |c'_X| \).

Definition 6.1 shows Zeller and Hildebrandt’s latest formulation of Delta Debugging [103]. The algorithm takes a test case \( c \) and a testing function \( \textit{test} \) as param-
etters. The test case represents the failure-inducing input to be minimized, while the testing function is used to determine whether an arbitrary test case triggers the original failure (by signaling \textit{fail} outcome, or $\mathbf{X}$) or not (by signaling \textit{pass} outcome, or $\mathbf{✓}$). (Additionally, according to its definition, a testing function may also signal \textit{unresolved} outcome, or $?$, but that outcome type is irrelevant to the algorithm.)

The algorithm first splits up the initial configuration into two subsets and checks whether any of them reproduces the original behaviour by itself. If none of them does, then the splitting ratio is doubled, and every new subset and their complements are evaluated by the test function, too. However, if any of the subsets or complements reproduces the expected behaviour, then that subset or complement will be used in the next iteration. The algorithm stops if every configuration of the finest possible splitting has been checked.

The result of the algorithm is a 1-minimal test case, where the definition of $n$-minimality is:

\textbf{Definition 6.2 (n-minimal test case)} A test case $c \subseteq c_\chi$ is $n$-minimal if \( \forall c' \subset c \cdot |c| - |c'| \leq n \Rightarrow (\text{test}(c') \neq \mathbf{X}) \) holds. Consequently, $c$ is 1-minimal if \( \forall \delta_i \in c \cdot c - \{\delta_i\} \neq \mathbf{X} \) holds, where $\delta$ denotes an arbitrary change between two configurations.

Informally, 1-minimality means, that with a certain partitioning granularity, there is not a single subset left that could be removed without loosing the original interestingness property, that is the original failure.

\section*{6.2 Hierarchical Delta Debugging}

In the most widespread use case, the elements of a configuration of \textit{ddmin} are the lines or characters of a text file and removal means actual deletion of parts from the content. However, in case of a complex input format, like Java, JavaScript, JSON, C++, etc., this kind of partitioning could result in many syntactically incorrect intermediate test cases, which lowers the performance of minimization by executing superfluous tests. Hierarchical Delta Debugging, or HDD for short, was one of the first attempts to eliminate the problem. HDD, as first defined in the paper of Misherghi and Su [65], takes Delta Debugging to the domain of hierarchical inputs (e.g., to XML DOM trees or parse trees) by iteratively applying \textit{ddmin} to levels of trees, progressing downwards from the root to the leaves. In that case, configurations are sets of tree nodes, and removal of a node causes the removal of the whole subtree rooted at that node. This approach can produce fewer invalid test cases than a structure-unaware \textit{ddmin}, thus result in fewer test attempts and faster minimization. It has to be noted that this HDD algorithm has no guarantees on minimality in general, however, the fixed-point iteration of the algorithm, called HDD*, is shown to yield 1-tree-minimal results.

Listing 6.1 shows the pseudocode of HDD (with some small changes in the presentation compared to the original formulation to keep its style in sync with the rest of this thesis). The algorithm begins with processing the top level of the tree, where level $= 0$. It counts and tags the nodes at the current level with the \textit{tagNodes} auxiliary function. If there are any nodes on the current level, then they are minimized by the \textit{ddmin} algorithm. \textit{ddmin} will return the minimal set of nodes, or subtree roots, that
must be kept for reproduction. Every node in the input configuration that has not been marked for keeping is deleted by the prune function.

However, it was soon realized that complete subtree removal was not necessarily the best pruning approach for several input formats, as so-pruned trees could easily generate syntactically invalid test cases and, therefore, cause many superfluous test attempts. Thus, in a subsequent work [66], Misharihi introduced the concept of the smallest allowable syntactic fragments, which should be used as replacements for nodes that are not kept by the Delta Debugging algorithm. As even the first paper on HDD considered trees built by using context-free grammars as the general, ideal case, they defined a recursive formula, Φ, to compute the smallest allowable syntactic fragments for context-free grammar non-terminals as follows:

\[
\Phi_0[e] = \begin{cases} 
  e & \text{if } e \in T \\
  \top & \text{if } e \in V 
\end{cases}
\]

\[
\Phi_i[e] = \min \left( \{ \Phi_{i-1}[e] \} \cup \{s|(e, r) \in P, s = \Phi_{i-1}[r_1] \Phi_{i-1}[r_2] \ldots \Phi_{i-1}[r_l] \} \right)
\]

where \( V \) is the set of nonterminals in the grammar, \( T \) is the set of terminals, \( P \) is the set of productions, and \( \top \) is a theoretical “top string” that is longer than any other string. Informally, terminals have a minimal-length string consisting only of themselves, while non-terminals concatenate the minimal-length string replacements of their derivations. With this change, they could ensure, that all the intermediate test cases during reduction are syntactically correct.

### 6.3 Related Work

One of the first and most influential works in the field of test case reduction is the Delta Debugging approach introduced by Zeller et al. [102, 36, 103], which can be applied to arbitrary input without having any a priori knowledge about the test case format. In exchange for this flexibility, it generates a large number of syntactically incorrect test cases that lowers its performance.

To lower the number of syntactically broken intermediate test cases, Miserghi and Su used context-free grammars to preprocess the test cases [65]. They converted the textual inputs to a tree representation and applied the ddmin algorithm to the levels of the tree. With this approach, called Hierarchical Delta Debugging, they could remove parts that aligned with syntactic unit boundaries. Although it substantially improved
DD both output quality and performance-wise, it still created syntactically incorrect test cases as it tried to remove every node even if that caused syntax errors. As an improvement, Miserghi analyzed the input grammar to decide which node can be completely removed and which should be replaced with a minimal but syntactically correct string [66]. This change guaranteed the intermediate test cases to be syntactically correct.

Tree-based test case reduction does not necessarily mean subtree removal. Bruno suggested to use hoisting as an alternative transformation in his framework called SIMP [8], which was designed to reduce database-related inputs. In every reduction step, SIMP tried to replace a node with a compatible descendant. In a follow-up work, they combined SIMP and Delta Debugging [68].

Chengnian et al. combine the above techniques in their Perses framework [87]. They are utilizing quantifiers and normalize the grammars by rewriting recursive rules to use quantified expressions. During reduction, they apply \textit{ddmin} to quantified nodes and hoisting to non-quantified ones. (Since this approach is language independent, it could have been a reasonable baseline for us in the following chapters, but we could not find the prototype tool publicly available.)

Herfert et al. [35] also combined subtree removal and hoisting in their Generalized Tree Reduction (GTR) algorithm but instead of analyzing a grammar to decide about the applicability of a certain transformation they learned this information from an existing test corpus. Regehr et al. also used transformations in their tool C-Reduce [81], which is used to reduce C/C++ sources, but they applied language-specific transformations based on the semantics obtained by Clang.

All the above works targeted textual failure-inducing inputs, but test case reduction has a much broader application area. Colin et al. [83] minimized faulty event sequences of distributed systems. Brummayer et al. [7] used Delta-Debugging in order to minimize failure-inducing SMT solver formulas. Clapp et al. [14] aimed at reducing faulty Android GUI event sequences with an improved \textit{ddmin} variant called ND3MIN. SimplyDroid [53] also targeted Android GUI event minimization but it represented input events as a hierarchy tree and applied HDD and two new variants for reduction. Delta Debugging was also used to reduce unit tests [55, 56] or even unit test suites [31].

The efficiency of reduction can be improved with additional information. The authors of Pneumbra [15] used dynamic tainting to identify failure-relevant inputs. Wang [95] optimized event trace minimization by specifying constraints on events and failures. Lin et al. [58] used lightweight user-feedback information to guide the recognition of suspicious traces.
“Deleted code is debugged code.”
— Jeff Sickel

7

Practical Improvements to DDMIN

7.1 Parallelization

Although the original definition of the ddmin, as shown in Definition 6.1, is more of a recursive math formula, all known implementations realize it as a sequential non-recursive procedure, e.g., the clean-room implementation of the Lithium [71] project or even the reference implementation [101] provided by Zeller. Therefore, we give an equivalent variant of $ddmin_2$ – the key component of the minimizing Delta Debugging algorithm – in a sequential pseudo-code of $ddmin_2^σ$ in Listing 7.1.

The first thing to notice is that although the implementations tend to use sequential loops to realize the “reduce to subset” and “reduce to complement” cases of $ddmin_2$, as exemplified in Listing 7.1, the potential for parallelization is there in the original formulation, since $\exists i \in \{1, \ldots, n\}$ does not specify how to find that existing $i$. Since $n$ can grow big for real inputs and test is often expected to be an expensive operation, we propose to make use of the parallelization potential and rewrite $ddmin_2$ to use parallel loops. The pseudo-code of the parallelized version is given as $ddmin_2^π$ in Listing 7.2.

In the pseudo-code we intentionally do not specify the implementation details of the parallel constructs, but we assume the following for correctness and maximum efficiency:

- First of all, assignments to a single variable are expected to be atomic. I.e., if $\text{found} = i$ gets parallelly executed in two different loop bodies (with values $i_1$ and $i_2$) then the value of $\text{found}$ is expected to become strictly either $i_1$ or $i_2$. (See lines 8 and 20.)

- The amount of parallelization in parallel forall is left for the implementation, but it is expected not to overload the system, i.e., start parallel loop bodies only until all computation cores are exhausted. (See lines 6 and 18.)

- parallel break is suggested to stop/abort all parallelly started loop bodies even if their computation has not finished yet (in a general case, this may cause computation results to be thrown away, variables left uninitialized, etc., which
Chapter 7. Practical Improvements to DDMIN

Listing 7.1. Non-recursive sequential pseudo-code of \texttt{ddmin}_2

\begin{verbatim}
procedure ddmin$^2_2(c', n)$
begin
  out:
  while true do begin
    (* reduce to subset *)
    forall i in 1..n do
      if test(\(\Delta_i(c', n)\)) = X then begin
        \(c' = \Delta_i(c', n)\)
        \(n = 2\)
        continue out
      end
    (* reduce to complement *)
    forall i in 1..n do
      if test(\(\nabla_i(c', n)\)) = X then begin
        \(c' = \nabla_i(c', n)\)
        \(n = \max(n - 1, 2)\)
        continue out
      end
    (* increase granularity *)
    if \(n < |c'|\) then begin
      \(n = \min(|c'|, 2n)\)
      continue out
    end
    (* done *)
  break out
end
return \(c'\)
end
\end{verbatim}

is always to be considered and taken care of, but it causes no issues in this algorithm). (See lines 9 and 21.)

7.2 Combination of the Reduce Cases

To improve the algorithm further, we have to take a look again at the original formulation of \texttt{ddmin}_2 in Definition 6.1. We can observe that although \texttt{ddmin}_2 seems to be given with a piecewise definition, the pieces are actually not independent but are to be considered one after the other, as mandated by the \texttt{else if} phrases. However, we can also observe that this sequentiality is \textit{not} necessary. There may be several \(\Delta_i\) and \(\nabla_i\) test cases that induce the original failure, we may choose any of them (i.e., we do not have to prefer subsets over complements) and we will still reach a 1-minimal solution at the end.

We cannot benefit from this observation as long as our implementation is sequential, but we propose to combine the two reduce cases, and test all subsets and complements in one step when parallelization is available. This way the algorithm does not have to wait until all subset tests finish but can start testing the complements as soon as computation cores become available. Listing 7.3 shows the pseudo-code of the algorithm variant \texttt{ddmin}_κ with the combined reduce cases.

The pseudo-code in the previous section – i.e., Listings 7.1 and 7.2 – are consistent with the original formal definition of Zeller and Hildebrandt, as shown in Algorithm 6.1. They specialize the formalization and differ from each other only in the way how the existency check \(\exists i \in \{1, \ldots, n\}\) is implemented (i.e., sequentially or parallelly).
However, the idea of combining the reduce cases as presented in Listing 7.3, although still yields 1-minimal results, deviates from the original formalization, since it explicitly merges the “reduce to subset” and “reduce to complement” checks. For the sake of comparability, we also give a formal definition of the algorithm variant in Zeller and Hildebrandt’s style in Definition 7.1.

7.3 De-prioritization or Omission of “Reduce to Subset”

Once we have observed that there is no strict need for sequentially investigating the two reduce cases of \( \text{ddmin}_2 \), we can also observe that the “reduce to subset” case is not even necessary for 1-minimality. It is a greedy attempt by the algorithm to achieve a significant reduction of the test case by removing all but one subsets in one step rather than removing them one by one in the “reduce to complement” case. However, there are several input formats where attempting to keep just the “middle” of a test case almost always gives a syntactically invalid input and thus cannot induce the original
Chapter 7. Practical Improvements to DDMIN

Listing 7.3. Parallel pseudo-code of \textit{ddmin}_2 with combined reduce cases

```plaintext
procedure \textit{ddmin}_2(c'_X, n)
begin
while true do begin
(* reduce to subset or complement *)
found = 0
parallel forall i in 1..2n do
  if 1 ≤ i ≤ n then
    if test(Δ(c'_X, n)) = X then begin
      found = i
      parallel break
    end
  else if n + 1 ≤ i ≤ 2n then
    if test(∇(c'_X, n) - i) = X then begin
      found = i
      parallel break
    end
  if 1 ≤ found ≤ n then begin
    c'_X = Δ(c'_X, found)
    n = 2
    continue
  end else if n + 1 ≤ found ≤ 2n then begin
    c'_X = ∇(c'_X, found - n)
    n = max(n - 1, 2)
    continue
  end
(* increase granularity *)
if n < |c'_X| then begin
  n = min(|c'_X|, 2n)
  continue
end
(* done *)
break
end
return c'_X
end
```

Definition 7.1 (Formal \textit{ddmin}_2 variant with combined reduce cases)

\[
\text{ddmin}_2^K(c'_X, n) = \begin{cases} 
  \text{ddmin}_2^K(\delta_i, n) & \text{if } \exists i \in \{1, \ldots, 2n\} \cdot \text{test}(\delta_i) = X \text{ ("reduce to subset or complement")}, \\
  \text{ddmin}_2^K(c'_X, \min(|c'_X|, 2n)) & \text{else if } n < |c'_X| \text{ ("increase granularity")}, \\
  c'_X & \text{otherwise ("done")}. 
\end{cases}
\]

where \(\delta_i = \begin{cases} 
  \Delta_i & \text{if } 1 \leq i \leq n \\
  \nabla_{1-n} & \text{if } n + 1 \leq i \leq 2n 
\end{cases}\) and \(\nu_i = \begin{cases} 
  2 & \text{if } 1 \leq i \leq n \\
  \max(n - 1, 2) & \text{if } n + 1 \leq i \leq 2n 
\end{cases}\).

failure. (C-like source files with the need for correct file and function headers and mandating properly paired curly braces are a typical example of such inputs.) For such input formats, the “reduce to complement” case may occur significantly more often, while the “reduce to subset” case perhaps not at all. Thus, we argue that it is worth experimenting with the reordering of the reduce cases, and also with the complete omission of the “reduce to subset” case, as it may be simply the waste of computation resources.

The idea of reordering the reduce cases can be applied to all previously introduced algorithm variants. However, presenting the modified algorithms again would be unnecessarily verbose for little benefit. Thus, we only refer to the algorithms shown earlier
Chapter 7. Practical Improvements to DDMIN

and mention those parts that need to be changed to test *complements first*:

- in Listing 7.1, lines 5–11 and 12–18 should be swapped,
- in Listing 7.2, swapping lines 4–15 and 16–27 achieves the same, while
- in Listing 7.3, lines 8–11 have to be swapped with lines 13–16, lines 18–20 with lines 22–24, and the subscript indices of $\Delta s$ and $\nabla s$ have to be updated so that the $-n$ element is applied to $\Delta s$.

The idea of omitting the “reduce to subset” case completely can reasonably be applied to the sequential and parallel variants only, as leaving the testing of subsets from the algorithm with combined reduce cases would be no different from the parallel variant. Thus, the changes to be applied for testing *complements only* are as follows:

- in Listing 7.1, lines 5–11 are to be deleted, while
- in Listing 7.2, lines 4–15 are superfluous.

### 7.4 Evaluation

During the investigation of the minimizing Delta Debugging algorithm, we created a prototype tool, named Picire [41], that implemented our proposed improvements. At the very beginning, it was based on Zeller’s public domain reference implementation but as new ideas got incorporated into it, it was rewritten several times until only traces of the original source remained. The tool was written in Python 3, and the parallel loop constructs of the algorithm variants were implemented based on Python’s multiprocessing module.

For the evaluation platform of our tool and our ideas, we have used a dual-socket Supermicro X9DRG-QF machine equipped with 2 Intel Xeon E5-2695 v2 (x86-64) CPUs clocked at 2.40 GHz and 64 GB DDR3 RAM at 1600 MHz. Each CPU had 12 cores and each core was hyper-threaded, which gave 24 physical cores and 48 processing threads (or logical CPUs seen by the kernel) in total. Cores is a hardware term that describes the number of independent central processing units in a single computing component (die or chip). Thread, or thread of execution, is a software term for the basic ordered sequence of instructions that can be passed through or processed by a single CPU core. Intel Hyper-Threading Technology (Intel HT Technology) delivers two processing threads per physical core. Highly threaded applications can get more work done in parallel, completing tasks sooner. The machine was running Ubuntu 14.04 with Linux kernel 3.13.0, and the native compiler was gcc 4.9.2.

All the approaches from the previous sections were tested with 10 examples and executed with 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48 and 64 levels of parallelism. As primary software targets to trigger failure in and minimize tests for, we have chosen two real-life web engines – from the WebKit [3] and Chromium [29] projects – for which a large number of test cases were available. The WebKit project was checked out from its official code repository at revision 192323 dated 2015-11-11 and built in debug configuration for its GTK+ port, i.e., external dependencies like UI elements were provided by the GTK+ project [32]. The SUT was the WebKitTestRunner program (executed with the --no-timeout command line option), a minimalistic web browser application used
Table 7.1. Test cases, sizes, and their interestingness condition.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Size</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium A (html)</td>
<td>31,626 bytes / 1,058 lines</td>
<td>ASSERTION FAILED: i &lt; size()</td>
</tr>
<tr>
<td>Chromium B (html)</td>
<td>34,323 bytes / 3,769 lines</td>
<td>ASSERTION FAILED: static_cast&lt;unsigned&gt;-&gt;offsetInNode &lt;= layoutTextFragment-&gt;start() + layoutTextFragment-&gt;fragmentLength()</td>
</tr>
<tr>
<td>Chromium C (html)</td>
<td>48,503 bytes / 1,706 lines</td>
<td>ASSERTION FAILED: !current.value() - isInheritedValue()</td>
</tr>
<tr>
<td>WebKit A (html)</td>
<td>23,364 bytes / 959 lines</td>
<td>ASSERTION FAILED: newLogicalTop = logicalTop</td>
</tr>
<tr>
<td>WebKit B (html)</td>
<td>30,417 bytes / 1,374 lines</td>
<td>ASSERTION FAILED: willBeComposited == needsToBeComposited(layer)</td>
</tr>
<tr>
<td>WebKit C (html)</td>
<td>36,051 bytes / 3,791 lines</td>
<td>ASSERTION FAILED: !needsStyleRecalc()</td>
</tr>
<tr>
<td>Example A (array)</td>
<td>8 elements</td>
<td>{5,8} ⊆ c ∧ (2 ∈ c ∨ 7 ∉ c)</td>
</tr>
<tr>
<td>Example B (array)</td>
<td>8 elements</td>
<td>{1,2,3,4,6,7,8} ⊆ c</td>
</tr>
<tr>
<td>Example C (array)</td>
<td>8 elements</td>
<td>{1,2,3,4,6,8} ⊆ c ∧ (5,7) ∉ c</td>
</tr>
<tr>
<td>Example D (array)</td>
<td>100 elements</td>
<td>(2x/0 ≤ x &lt; 50) ⊆ c</td>
</tr>
</tbody>
</table>

Table 7.2. Size of 1-minimal test cases in lines.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Sequential</th>
<th>Parallel</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium A</td>
<td>31 (2.93%)</td>
<td>31–34 (2.93–3.21%)</td>
<td>31–34 (2.93–3.21%)</td>
</tr>
<tr>
<td>Chromium B</td>
<td>34 (0.90%)</td>
<td>34–37 (0.90–0.98%)</td>
<td>34–37 (0.90–0.98%)</td>
</tr>
<tr>
<td>Chromium C</td>
<td>9 (0.53%)</td>
<td>9 (0.53%)</td>
<td>9 (0.53%)</td>
</tr>
<tr>
<td>WebKit A</td>
<td>11 (1.15%)</td>
<td>11 (1.15%)</td>
<td>11 (1.15%)</td>
</tr>
<tr>
<td>WebKit B</td>
<td>21 (1.53%)</td>
<td>19–23 (1.38–1.67%)</td>
<td>19–23 (1.38–1.67%)</td>
</tr>
<tr>
<td>WebKit C</td>
<td>46 (1.21%)</td>
<td>46 (1.21%)</td>
<td>46 (1.21%)</td>
</tr>
<tr>
<td>Example A</td>
<td>2 (25.00%)</td>
<td>2 (25.00%)</td>
<td>2 (25.00%)</td>
</tr>
<tr>
<td>Example B</td>
<td>8 (100.00%)</td>
<td>8 (100.00%)</td>
<td>8 (100.00%)</td>
</tr>
<tr>
<td>Example C</td>
<td>6 (75.00%)</td>
<td>6 (75.00%)</td>
<td>6 (75.00%)</td>
</tr>
<tr>
<td>Example D</td>
<td>50 (50.00%)</td>
<td>50 (50.00%)</td>
<td>50 (50.00%)</td>
</tr>
</tbody>
</table>

in the testing of layout correctness of the project. The Chromium project was checked out from its official code repository at revision hash 6958b6e dated 2015-11-29 and also built in debug configuration. The SUT from that project was the content_shell tool (executed with the --single-process --no-sandbox --run-layout-test options), also a minimal web browser. As a secondary target, we have implemented a mock SUT, a parametrized tester function that can unit test the algorithm implementations for correctness by working on abstract arrays as inputs and allowing an explicit control of what is accepted as failure-inducing or passing, and also allows experimenting with various test execution times (since the time required for the evaluation of the failure condition is negligible and the tester can lengthen its running time arbitrarily). The point of using artificial tests beside the real ones are based on 2 major differences between them. First, real life test cases are flexible in the point of the failure condition – which means that multiple different but mostly similar subsets could reproduce the expected failure. That is, there may exist multiple different 1-minimal outcomes. This can have an effect both on the size of the final outcome and also on the running time of reduction in multiprocess mode. Contrarily, artificial test cases have exactly one 1-minimal result. For this reason, their results will be much more stable since they leave less room for randomness.

For each of the two browser engine targets, we have selected 3 fuzzer-generated test cases – HTML with a mixture of SVG, CSS, and JavaScript – that triggered various assertion failures in the code. The average size of the tests was around 2000 lines, with the shortest test case being 959 lines long and the longest one growing up to
3769 lines. For the mock SUT, we have used 4 manually crafted test cases, which consisted of an array of numbers and a condition to decide about the interestingness of a certain subset. Three of them were small tests imported from Zeller’s reference examples, while the larger fourth (with 100 elements) was specially crafted by us for a corner case where the only 1-minimal subset contains exactly the odd elements. As a consequence, all approaches are only able to remove anything at the finest splitting. Table 7.1 details the sizes and the interestingness conditions of all test cases.

To always have a clear picture about the effect of a certain variation, we will evaluate the parallelization and non-parallelization-based approaches (subset-first, complement-first, complement-only) separately.

In the evaluation of the algorithm variants that we have proposed in the previous sections, our first step was to check the sizes of the minimized test cases. The reasons for the investigation were two-fold: first, as the manually crafted test cases have exactly one 1-minimal subset, they acted as a sanity check. Second, as 1-minimal test cases are not necessarily unique in general, we wanted to see how the algorithm variants affect the size of the result in practice, i.e., on the real browser test cases.

We have examined the effect of the reduce step variations (“subsets first”, “complements first”, and “complements only”; as described in Section 7.3) on the algorithm and the effect of the two parallelization approaches (“parallel” and “combined”; as described in Sections 7.1 and 7.2) independently. It turned out that all reduce variants of the sequential algorithm gave exactly the same result, not only for the examples tests but for the real browser cases as well (see the second column of Table 7.2). Detailed investigations have revealed that “reduce to subset” is a very rare step in practice (it happened in the first iteration only, with \( n = 2 \), when subsets and complements are equal anyway) and because of the sequential nature of the algorithm, the “reduce to complement” steps were taken in the same order by all algorithm variants.

Figure 7.1 shows the progress of the Chromium B test’s reduction over the iterations. The diagram contains information from all three approaches, however only one is visible since they overlap with each other as it was discussed in the previous paragraph. The rest of test cases result in similar plots so they are not presented here.
Chapter 7. Practical Improvements to DDMIN

As the next step, we inspected the influence of parallelism on the output sizes. For this, we executed the browser tests with different levels of parallelism (1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48 and 64 processes respectively) both in parallel and combined modes. The output sizes are shown in the third and fourth columns of Table 7.2 and in Figure 7.2.

The numbers in the table show differences in the output sizes. This is explicable by the fact that there are multiple ways of reduction and that the parallel approaches also chose greedily between them. That is, the first one seen will be used in the next iteration – instead of the sequentially first. The deviations mean 0-4 line differences. To evaluate these numbers correctly, it is worth to see them compared to their original size. Figure 7.3 shows the same results but normalized to the input’s size. They show that the differences are always less than 1%, which means that the effect of parallelization on the output sizes is negligible.
In our second experiment, we have investigated the effect of the algorithm variants on the number of iterations and test evaluations needed to reach 1-minimality. First, Figure 7.4 exemplifies the effect of varying the reduce steps in the sequential algorithm on one selected test case, then Table 7.3 gives the results of this experiment on all test cases.

Thus, in Figure 7.4, we show details of minimizations of the WebKit A test case. Subfigure (a) plots how the size of the test case gets reduced over iterations of the sequential algorithm. The remaining subfigures (b–d) tell the granularity of each iteration as well as the tests evaluated per iteration (where cache hits are distinguished from real test evaluations but stacked on each other). The following notes may help the interpretation of the bar charts: whenever the cumulative height of the gray and white boxes (i.e., evaluated tests and cache hits) is lower than the black one (i.e., granularity), then the first reduce step was chosen; if the gray and white boxes are somewhat higher
Figure 7.4. Minimization of the WebKit A test case with variations of the sequential algorithm. In subfigures (b–d), black bars denote the granularity in each iteration – i.e., the number of subsets the test case was split to –, gray bars denote how many times the test function was really evaluated, while white bars signal how many times the result of test function was retrieved from cache.
Table 7.3. The number of iterations and test evaluations needed to reach 1-minimality. First numbers or intervals in each column stand for evaluated tests, the second numbers or intervals in parentheses with + prefix stand for cache hits, while the third numbers or intervals in brackets stand for iterations performed.

(a) Comparison on the effects of reduce step variations (performing all tests sequentially).

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Subsets First</th>
<th>Complements First</th>
<th>Complements Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium A</td>
<td>531 (+1589) [76]</td>
<td>466 (+54) [76]</td>
<td>343 (+24) [76]</td>
</tr>
<tr>
<td>Chromium B</td>
<td>560 (+1931) [83]</td>
<td>489 (+9) [83]</td>
<td>356 (+3) [83]</td>
</tr>
<tr>
<td>Chromium C</td>
<td>263 (+388) [56]</td>
<td>218 (+5) [56]</td>
<td>159 (+2) [56]</td>
</tr>
<tr>
<td>WebKit A</td>
<td>265 (+479) [53]</td>
<td>222 (+7) [53]</td>
<td>161 (+3) [53]</td>
</tr>
<tr>
<td>WebKit B</td>
<td>503 (+1674) [84]</td>
<td>430 (+21) [84]</td>
<td>319 (+7) [84]</td>
</tr>
<tr>
<td>WebKit C</td>
<td>831 (+5127) [129]</td>
<td>714 (+14) [129]</td>
<td>509 (+5) [129]</td>
</tr>
<tr>
<td>Example A</td>
<td>22 (+22) [8]</td>
<td>17 (+5) [8]</td>
<td>14 (+1) [8]</td>
</tr>
<tr>
<td>Example B</td>
<td>26 (+2) [3]</td>
<td>26 (+2) [3]</td>
<td>14 (+0) [3]</td>
</tr>
<tr>
<td>Example D</td>
<td>472 (+3237) [57]</td>
<td>422 (+16) [57]</td>
<td>276 (+0) [57]</td>
</tr>
</tbody>
</table>

(b) Comparison on the effects of parallelization (testing subsets first).

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Sequential</th>
<th>Parallel</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium A</td>
<td>531 (+1589) [76]</td>
<td>531–2476 (+1568–2389) [76–89]</td>
<td>531–2933 (+1574–2386) [76–89]</td>
</tr>
<tr>
<td>Chromium C</td>
<td>263 (+388) [56]</td>
<td>263–589 (+388–394) [56]</td>
<td>263–958 (+22–388) [56]</td>
</tr>
<tr>
<td>WebKit A</td>
<td>265 (+479) [53]</td>
<td>265–675 (+477–486) [53]</td>
<td>265–708 (+452–483) [53]</td>
</tr>
<tr>
<td>WebKit B</td>
<td>503 (+1674) [84]</td>
<td>503–2116 (+1668–1771) [84–85]</td>
<td>503–2132 (+1623–1762) [84–85]</td>
</tr>
<tr>
<td>WebKit C</td>
<td>831 (+5127) [129]</td>
<td>831–5670 (+5127–5136) [129]</td>
<td>831–5697 (+5115–5132) [129]</td>
</tr>
<tr>
<td>Example B</td>
<td>26 (+2) [3]</td>
<td>26 (+2) [3]</td>
<td>26–28 (+0–2) [3]</td>
</tr>
<tr>
<td>Example D</td>
<td>472 (+3237) [57]</td>
<td>472–3398 (+3237–3251) [57]</td>
<td>472–3440 (+3009–3249) [57]</td>
</tr>
</tbody>
</table>

than the black bar, then the second reduce step was taken; but if twice as many tests were evaluated as the granularity then the iteration was unable to reduce the test case. Based on these notes, we can clearly see in Figure 7.4(b) that with the original “subsets first” approach, the first “reduce to subset” step was never taken (but tried over and over again, and only a cache could help prevent their excessive re-evaluation). Figures 7.4(c–d) show how “complements first” and “complements only” performed. The most outstanding difference is the almost complete lack of cache hits – which is reasonable as the cache is mostly utilized by subset tests.

In Table 7.3(a), we can observe the effects of the variation of the reduce steps in the sequential algorithm on all test cases. Although the number of iterations never changed, the number of test evaluations dropped considerably everywhere. Even if we used caching\(^1\) of test results, the “complements first” variant saved 12–17% of test evaluations on real browser test cases, while “complements only” saved 35–40%. The test cases of the mock SUT show a bit more scattered results with the “complements first” variant (0–23% reduction in test evaluations), but “complements only” achieves a similar 36–41%. As test results are re-used heavily by the original “subsets first”

---

\(^1\)In the context of Delta Debugging, caching is the memorization and reuse of the outcomes of the testing function on already investigated test cases. The caching mechanism is not incorporated in the definition of the algorithm, since it can be left for the test function to realize.
Figure 7.5. Running time of the minimization of the WebKit A test case (with all algorithm variants, as the level of parallelization increases). Gray lines stand for sequential algorithms, thin black lines represent parallel variants, while thick black lines plot the “combined” approach. Solid, dotted, and dashed patterns represent the “subsets first”, “complements first”, and “complements only” reduce step variations, respectively. (Note 1: the “complements only” variant is the same for the two parallelization approaches. Note 2: axes are logarithmic to ease the separation of short running times.)

approach, if we would also interpret those cache hits as test evaluations then that could mean a dramatic 90% reduction in some cases. Table 7.3(b) contains the results of parallelizations – yet again with 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, and 64 parallel loop bodies allowed. These results might look disappointing or controversial at first sight as the number of evaluated tests increased, sometimes by a factor of 6 or even more. However, as we will soon see, this increase is the natural result of parallelization and is not mirrored in the running time of the minimizations.

In our third and last experiment, we have investigated the effect of the algorithm variants on the running time of minimization. Again, we have selected the WebKit A test case to exemplify the results. It is visible from Figure 7.5 that even if parallelization is not available, varying the reduce steps could decrease the running time by 14% and 35%, well aligned with the results presented in Table 7.3(a). However, when we enabled parallelization (e.g., 64 parallel loop bodies), we could reduce the running time by 73–75%. Unfortunately, by combining parallelization and reduce step variants, we could not simply multiply these reduction factors. E.g., with 64-fold parallelization, the effect of “complements first” became a 2–5% gain in running time, and the effect of “complements only” also shrank to 7–15%. Nevertheless, that is still a gain, the combined parallel loops with only complements tests performed gave the best results and reduced the time of the minimization to 72 seconds from the original 316, achieving an impressive 77% reduction.

However, we did not only focus on the WebKit A test case, but we have taken all 6 real browser tests, plus parameterized the 4 artificial test cases to run for 0 seconds, 1 second, and randomly between 1 and 3 seconds. That gave 18 test cases in practice, for $3 \times 3$ algorithm variants, on 12 different levels of parallelization. Even though we
Table 7.4. Algorithm variants giving the highest reduction in running time of minimization.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Best Algorithm (Parallelization Level)</th>
<th>Best/Original Time</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium A</td>
<td>Parallel Complements-Only (24)</td>
<td>231s / 904s</td>
<td>74.42%</td>
</tr>
<tr>
<td>Chromium B</td>
<td>Parallel Complements-Only (16)</td>
<td>180s / 890s</td>
<td>79.78%</td>
</tr>
<tr>
<td>Chromium C</td>
<td>Parallel Complements-Only (8)</td>
<td>102s / 413s</td>
<td>75.28%</td>
</tr>
<tr>
<td>WebKit A</td>
<td>Parallel Complements-Only (12)</td>
<td>70s / 316s</td>
<td>77.63%</td>
</tr>
<tr>
<td>WebKit B</td>
<td>Parallel Complements-Only (12)</td>
<td>127s / 584s</td>
<td>78.26%</td>
</tr>
<tr>
<td>WebKit C</td>
<td>Parallel Complements-Only (12)</td>
<td>192s / 915s</td>
<td>78.97%</td>
</tr>
<tr>
<td>Example A (0s)</td>
<td>Sequential Complements-Only (1)</td>
<td>0.0020s / 0.0023s</td>
<td>13.79%</td>
</tr>
<tr>
<td>Example A (1s)</td>
<td>Combined Subsets-First (64)</td>
<td>7s / 22s</td>
<td>67.79%</td>
</tr>
<tr>
<td>Example A (1–3s)</td>
<td>Combined Subsets-First (32)</td>
<td>11s / 45s</td>
<td>75.34%</td>
</tr>
<tr>
<td>Example B (0s)</td>
<td>Sequential Complements-Only (1)</td>
<td>0.0011s / 0.0014s</td>
<td>23.58%</td>
</tr>
<tr>
<td>Example B (1s)</td>
<td>Parallel Complements-Only (16)</td>
<td>3s / 26s</td>
<td>88.29%</td>
</tr>
<tr>
<td>Example B (1–3s)</td>
<td>Parallel Complements-Only (16)</td>
<td>7s / 50s</td>
<td>85.91%</td>
</tr>
<tr>
<td>Example C (0s)</td>
<td>Sequential Complements-Only (1)</td>
<td>0.0016s / 0.0020s</td>
<td>20.80%</td>
</tr>
<tr>
<td>Example C (1s)</td>
<td>Parallel Complements-Only (24)</td>
<td>5s / 30s</td>
<td>83.13%</td>
</tr>
<tr>
<td>Example C (1–3s)</td>
<td>Parallel Complements-Only (16)</td>
<td>8s / 53s</td>
<td>84.78%</td>
</tr>
<tr>
<td>Example D (0s)</td>
<td>Sequential Complements-Only (1)</td>
<td>0.0436s / 0.0762s</td>
<td>42.76%</td>
</tr>
<tr>
<td>Example D (1s)</td>
<td>Parallel Complements-Only (64)</td>
<td>58s / 472s</td>
<td>87.69%</td>
</tr>
<tr>
<td>Example D (1–3s)</td>
<td>Parallel Complements-Only (64)</td>
<td>79s / 930s</td>
<td>91.48%</td>
</tr>
</tbody>
</table>

skipped the re-evaluation of duplicates (e.g., “complements first” is the same for “parallel” and “combined” variants, or sequential algorithms run unchanged independently of the available computation cores), we got the results of 1098 minimization executions, all of which would be hard to picture within the limits of this thesis. Thus, we present only the best results for each test case in Table 7.4.

The results clearly show that for most of the cases (12 of 18) the “parallel complements only” variant gave the highest running time reduction – as expected. Perhaps it is worth discussing those elements of the table, which are a bit less expected: first of all, it might be surprising that parallelization did not always work best at the highest level used (64). However, we have to recall that the machine used for evaluation has 24 real cores only (and 48 logical CPUs seen by the kernel, but those are the result of hyper-threading), and it should also be considered that the browser SUTs are inherently multi-threaded or even multi-process, thus even a single instance may occupy more than one core. It may also come as a surprise that a sequential algorithm turned out to perform best for some artificial test cases. These were the cases when the evaluation of the interestingness of a test was so quick that the overhead of the parallelization implementation became a burden. (In practice, test implementations rarely run so fast.)

As a summary, we can conclude that all proposed variants of the Delta Debugging algorithm yielded a measurable speed-up, with best combinations gaining cca. 75–80% of the running time on real test cases, i.e., needing only one-fourth to one-fifth of the original execution time to reach 1-minimal results. Moreover, on manually crafted test cases, even higher running time reduction could be observed.

### 7.5 Conclusions

In this chapter, we have taken a close look at the minimizing Delta Debugging algorithm which is the most widespread solution for syntax unaware automatic test case
reduction tasks. Our original motivation was to enable the algorithm to utilize the parallelization capabilities of multi-core and multi-processor computers, but as we analyzed it, we found further enhancement possibilities. We recognised the parallelization potential implicitly given in the existence checks of the original definition of the algorithm (although never exploited by its authors), but we have also pointed out that the same original definition unnecessarily prescribed sequentiality elsewhere, for its “reduce to subset” and “reduce to complement” steps. Moreover, we have argued that the “reduce to subset” step of the original algorithm may even be omitted completely without losing its key property of resulting 1-minimal test cases, but being more effective for some types of inputs. Each observation has led to an improved algorithm variant, all of which are given in pseudo-code. Finally, we implemented a prototype tool called Picire and presented an experiment conducted on 4 artificial test cases and on 2 wide-spread browser engines with 3 real test cases each. All test cases were minimized with all presented algorithm variants and with 12 different levels of parallelization (ranging from single-core execution – i.e., no parallelization – to 64-fold parallelization on a computer with 48 virtual cores). The results of the 1098 successfully executed test case minimizations prove that all presented improvements to the Delta Debugging algorithm achieved performance improvements, with the best variants reducing the running time on real test cases significantly, by cca. 75–80%.
“If debugging is the process of removing software bugs, then programming must be the process of putting them in.”

— Edsger Dijkstra

8

HDD on Extended Context-Free Grammar-based Parse Trees

8.1 Motivation

Even if Mishergi and Su’s HDD algorithm [65, 66] is general – i.e., no restriction is placed on how the tree it is working on is built –, its discussion and evaluation, and especially the minimal replacement string computation are greatly tied to standard context-free grammars. Unfortunately, as even the original authors note, standard context-free grammars use recursion to represent lists, which yields heavily unbalanced trees. This property does not only increase the number of iterations in HDD but also has an effect on the size of the reduced test case.

The HDD algorithm has only one reference implementation available [64] written mostly in Python 2 and utilizing Zeller’s DD routine [101], with flex and bison-generated native lexer and parser routines bound to the Python code with the help of the SWIG interface compiler, and minimal parser rule computation supported by an ANTLRv2-based algorithm. This quite heterogeneous component list – with some very outdated elements – not only makes the dependencies of the project hard to fulfill but also complicates its usage. As depicted in Figure 8.1, making the project operational requires several preparatory steps involving flex and bison invocations and the building of a shared library, which is unusual (or at least inconvenient) in the case of a Python project. Moreover, when the build step finishes, the user of HDD also has to provide a module (e.g., write a program with a command line interface) that connects and drives all components of the system.

Finally, although the project is flex and bison-based, its grammar file format is – strictly speaking – incompatible with those tools. Albeit the parser rules are to be defined in bison format, they have to be combined into a single file with lexer rules. Moreover, lexer rules must be annotated with their manually computed minimal replacement strings. All this means that even if flex and bison grammar rules are available for a given input format, they cannot be put to use without potentially considerable efforts of preprocessing.
8.2 The Modernized HDD

Now, we discuss how to deal with the shortcomings identified in the previous section. First of all, we argue that instead of standard context-free grammars, extended context-free grammars (ECFGs) should be used for parsing inputs and building the input trees. Extended context-free grammars allow the right-hand side of rules to be regular expressions over terminals and non-terminals, i.e., alternation operators, groupings, and quantifiers (\(?\), \(*\), \(+\)) may appear. While extended context-free grammars describe exactly the context-free languages, just like standard context-free grammars do, the quantifiers allow the omission of recursive rules for list-like structures, and this results in much better balanced parse trees.

Using ECFGs requires the adaptation of the minimal replacement string computation algorithm to production rules with regular expression right-hand sides. We do not give the formal definition here, but informally mention that the algorithm has to consider the right-hand side of every rule as an expression tree where internal nodes represent concatenation, alternation and repetition (quantifiers), while the leaves are the terminal and non-terminal symbols of the ECFG. By recursively walking such an expression tree, the minimal string for a concatenation node can be calculated as the concatenation of the minimal strings of its children, for an alternation node as the minimum of the minimal strings of its children, for a + quantifier as the minimal string of its single child, and for * and ? quantifiers as the empty string. With this extension, the original fixed-point iteration algorithm can handle the proposed extended context-free grammars as well. A fortunate side-effect of this extension is that because the tokens of parser grammars given as ECFG tend to be given with regular expressions over characters themselves, the extended minimal replacement calculation algorithm can be used for lexer grammars as well.

Another important effect of the use of ECFGs is that when a node in a parse tree is not kept by DD, its smallest allowed syntactic fragment is not necessarily the minimal replacement string calculated for the corresponding terminal or non-terminal grammar symbol, but it can be the empty string when that particular parse tree node corresponds to a quantified part of the production rule of its parent node.

In a Python 3 package named Picireny [42], we have implemented the above outlined algorithmic changes, making use of Picire [45], that was introduced in the previous chapter, and ANTLRv4 [74], a parser generator tool that allows the specification of grammars in extended context-free form. In addition to the changes in the algorithms, we took care to avoid the usability issues of the original implementation. First, we have made the system as homogeneous as possible, i.e., except for the Java-based ANTLR command line tool, all components are purely Python 3-based (even the lexers and
parsers generated by ANTLR). Thanks to the dynamic nature of the Python language, no preceding build step is needed when an input needs to be reduced, as Picireny can invoke ANTLR on its grammar and dynamically load the generated lexer and parser during runtime. Moreover, a convenient command line interface is included in the package along with the API to facilitate its use in practice. The overview of the architecture of Picireny is presented in Figure 8.2.

As we had serious struggles with the original implementation when it came to the grammar specifications, we took extra care to stay format-conforming in Picireny. As we will also show in Section 8.3, our system can be used by simply downloading existing grammars from the public repository of ANTLR without any further need for manual edits, file concatenations, etc. As both lexer and parser rule replacements can be automatically calculated now, the manual specification of lexer rule replacement strings is not necessary anymore. However, if users still wish to override automation, they can do so by providing a separate replacement description file.

Last but not least, we mention one more useful technicality. Sometimes a single grammar is not enough to describe the structure of an input. E.g., complex web pages composed of various HTML, style and script parts cannot be correctly (or completely) parsed by an HTML grammar only. Even in the best case, the HTML lexer would recognize all the styles and scripts as single tokens, losing their internal structures and thus losing granularity as well when it comes to the HDD algorithm. Similar examples are regular expressions, URLs, paths, macros, JSON strings, etc. that are included in various host languages. To deal with such inputs, we have prepared Picireny to support so-called island grammars and allow the user to define those lexer tokens of the host grammar that incorporate structured content adhering to different rules. After building trees from these tokens by the island parsers, their roots are plugged into the main parse tree in place of the original terminal node.

8.3 Evaluation

In this section, we present experimental results obtained during the evaluation of our modernized implementation, Picireny. Our platform during the experiments was a
quad-core Intel Core i5 CPU clocked at 2.80GHz equipped with 20GB RAM. The machine was running Ubuntu 15.10 with Linux kernel 4.2.0.

Our primary goal in the evaluation was to compare the results of our tool with the reference implementation of HDD. First, to have a fair comparison, we took the only publicly available test case that was used for the evaluation of HDD by its authors. The test case, a Java source code, contains an expression that ends in a division by zero. The interestingness criterion for reduction is based on this division, but it ignores semantic errors. Informally, this means that a test is considered failing if it contains the string ‘/0’ but it is not actually passed to the Java compiler, thus class, method, and variable identifiers may be replaced incorrectly. The original archive contained all the needed resources along the test input, e.g., the bison grammar and the minimal replacements of the lexer tokens were available. As the official ANTLR grammar repositories [1] also contain a grammar for Java, the test case could be executed with both tools. Listing 8.1 contains the reduced test case produced by HDD, while Listing 8.2 is the output of Picireny.

Although the results are similar, we can still observe some differences. First, Listing 8.1 contains everything in a single line, while Listing 8.2 is split into multiple lines, better reflecting the layout of the input file. This comes from a difference in the unparsing methods, i.e., how kept nodes of a tree are written to an output file (while HDD joins every kept token with a single space character, Picireny tracks the type of whitespaces (if any) that separate tokens in the original input and uses that knowledge in the unparsing step). Second, the result of HDD contains many extra semicolons, which are absent from the output of Picireny. Ultimately, this is rooted in the different types of context-free grammars (standard and extended) used by the two approaches and the parse trees built from them. Listings 8.3 and 8.4 show excerpts of parser grammars of the Java language. In both grammars, the block_statements rule is responsible for recognizing the contents of a compound statement. However, while it is expressed as a left-recursive rule for bison, it is a non-terminal that appears *-quantified in the right-hand side of the block rule in the ANTLR version. Therefore, although the minimal replacement string for block_statements is a single semicolon in both grammar variants, the extended-context-free grammar-based Picireny can replace the non-kept *-quantified subtrees with the empty string while HDD using the standard context-free grammar concepts is obliged to keep the semicolons. (Third, we may observe that the denominator of the division by zero is different in the two results. However, this is
nothing algorithm-specific. Because of differences in grammar structures and in the order of rules, the two approaches find different minimal replacement strings for the left-hand side of the division expression; but both approaches find a single character replacement.)

In addition to inspecting the outputs, we compared some metrics of the two reduction processes, too. We have measured the size of the inputs and outputs in various ways, the number of executed tests and the overall time needed by the tools to perform the reductions. We have investigated both the non-structured size metrics (i.e., the number of lines, bytes, and non-whitespace characters) of the input and output files, and as both tools build trees in a similar way, we have measured the height of the trees, and the number of terminals (leaves) and non-terminals (inner-nodes) in the trees as well.

The results are presented in Table 8.1. As the Output source section and the output listings show, there is not much difference between them. However, in the Input tree category, the dimensions of the built trees are different. The HDD tree is more than 7 times higher than the Picireny version, consequently it contains much more non-terminal nodes. This variance is a straight consequence of the way repetitions are expressed by the grammars.

The Output tree section of the table shows details about the final trees. Terminal nodes and Non-terminal nodes denote the number of kept nodes, while the fourth entry, the Non-empty non-kept nodes, gives the number of nodes (either terminals or non-terminals), that are marked to be removed but have a non-empty minimal replacement string. This data has to be taken into consideration since these nodes also contribute to the final output. As it turns out that both implementations keep only a single terminal node in their output tree, the difference in the number of non-empty non-kept nodes becomes highly important.

As the next evaluation step, we took failing test cases from real life. The chosen
tests caused assertion failures in the rendering engine of Google Chrome at revision 402879. For the evaluation, we used a debug version of the `content_shell` minimal browser.

Our first test case was an HTML file with styles and scripts included. However, the failure was only caused by an unexpected structure of three HTML tags although the size of the whole test case was more than 30KB. To perform the reductions and the comparison, we ran both HDD and Picireny with HTML grammars only. Since this target was not provided with the published HDD implementation, we had to perform the preparatory steps ourselves. This step included the quest for an HTML bison grammar and the refactoring of an HTML flex grammar to suite the needs of HDD. To run Picireny, again, we only had to download the HTML grammar from the official repository. The details of the various runs are shown in Table 8.2.

The input test contains 662 lines and more than 30KB data. The built trees show the same pattern as observed at the Java example, i.e., the tree built by HDD is much deeper and hence it contains more inner nodes. Considering the outcomes, the result of HDD with 478 bytes is two times larger than Picireny’s 238 bytes, even if they only correspond to 1.5% and 0.7% of the original input. The reason behind the difference is the difference in the grammar structures, again, although not completely in the same way as for the Java example. Unlike the ANTLR variant, the bison version of the HTML grammar does not require the opening and closing tags to be paired, ending up in independent removals and leaving several closing tags behind. Another interesting observation is the number of executed tests. In this case, HDD outperformed Picireny since it executed 60 tests less. The grammars explain this difference too, but not in a way that would make any of them superior to the other. Representing the same input with trees of highly different shapes will naturally place the units of the input (and so the failing-inducing parts) to different places. It would be possible to tweak the input to have the failure-inducing part higher up or deeper down the tree for both grammars, and thus force them to reach the final result sooner or later, respectively.

For the sake of a fair comparison, Picireny was executed in single-process mode in the previous tests even though it is able to execute tests in parallel (as it is built on
Chapter 8. HDD on Extended Context-Free Grammar-based Parse Trees

Table 8.2. HTML example

<table>
<thead>
<tr>
<th></th>
<th>HDD</th>
<th>Picireny</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input source</td>
<td>Lines</td>
<td>662</td>
</tr>
<tr>
<td></td>
<td>Bytes</td>
<td>31173</td>
</tr>
<tr>
<td></td>
<td>Non-ws chars</td>
<td>27660</td>
</tr>
<tr>
<td>Output source</td>
<td>Lines</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Bytes</td>
<td>478</td>
</tr>
<tr>
<td></td>
<td>Non-ws chars</td>
<td>319</td>
</tr>
<tr>
<td>Input tree</td>
<td>Height</td>
<td>2934</td>
</tr>
<tr>
<td></td>
<td>Terminal nodes</td>
<td>3806</td>
</tr>
<tr>
<td></td>
<td>Non-terminal nodes</td>
<td>3638</td>
</tr>
<tr>
<td>Output tree</td>
<td>Height</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>Terminal nodes</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Non-terminal nodes</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>Non-empty non-kept nodes</td>
<td>59</td>
</tr>
<tr>
<td>Executed tests</td>
<td></td>
<td>203</td>
</tr>
<tr>
<td>Time (s)</td>
<td></td>
<td>1872</td>
</tr>
</tbody>
</table>

the parallel DD implementation introduced in the previous chapter). However, when exploiting the parallelization capability and running 4 tests in parallel, we could reduce the running time by 31%, to 1810 seconds, and achieve the same reduced output. With this speed-up Picireny could outperform HDD.

In our third experiment, we wanted to evaluate the island grammar support of Picireny (and as a fortunate side-effect, we also discovered its error tolerance). The chosen test was also written in HTML and contained style (CSS) and script (JavaScript) definitions. However, in this case not only the HTML, but also the style and script sources were contributing to the failure. Using only one grammar would have recognized these inline fragments as single tokens and since they could not have been removed as a single unit, they would not have been changed – i.e., reduced – at all. Moreover, the test case was generated randomly and it contained syntax errors, which made it impossible to parse and reduce it with HDD, as the used grammar expected the original test case to be syntactically correct. Because of the above issues, we could not manage to execute HDD for this test case. Another comparison baseline could have been a simple line-based DD reducer, but because of the random generated nature of the test, most of the HTML part was on a single line, leaving no chance for DD to reduce it and also making DD-based comparisons pointless.

Since we could not compare Picireny to any other tool in this case, we feel it is fair to use parallelism and other improvements. Beside using 4-fold parallelism, we have configured the underlying DD algorithm according to the suggestions in the previous chapter, and so we ran only complement tests. The minimized test case is shown in Listings 8.5 and the metrics of the reduction are presented in Table 8.3. As the figures show, the test case was squeezed from 29610 bytes to 359 bytes, which is only 1% of the original file. In this evaluation, we had to run 1014 test cases in an hour.
8.4 Conclusions

In this chapter, we have investigated the decade-old HDD algorithm and its implementation, and proposed improvements to address their shortcomings. We have found that although HDD itself is not strictly dependent on how the tree it is working on is built, both its discussion and its reference implementation are tied to standard context-free grammars. We have argued that using extended context-free grammars for HDD is beneficial in several ways, as the use of quantifiers in the right-hand side of grammar rules instead of recursive rules yields more balanced parse trees and enables an improved strategy for “smallest allowed syntactic fragment replacement”, which in turn can result in a smaller output. In addition to the grammar-related proposals, we have pointed out practical issues that hinder the use of the reference implementation. All the enhancement ideas and usability improvements are embodied in a modernized HDD implementation tool called Picireny.

The new implementation

- uses ANTLRv4 grammars supporting extended context-free grammar notation,
• is more homogeneous than its predecessor (containing only two language dependencies, Python and Java),

• implements minimal string replacement calculation for lexer tokens as well,

• supports island grammars for inputs with multi-language structures,

• uses standard grammar file formats and has easy access to already available grammar definitions,

and thus – because of its ease of use – it can hopefully help in spreading the use of HDD in practice.

Experimental evaluation of Picireny supports the grammar-related ideas outlined in this chapter: its reduced outputs are significantly smaller (by cca. 25–40%) on the investigated test cases than those produced by the original HDD implementation using standard context-free grammars (if counting the non-whitespace characters in the result, which metric is perhaps the least biased by pretty printing strategies or grammar representations).
“I don’t know where you are, I don’t know how you work, but, I will find you and I will fix you.”
— Ankit Pithadiya

9

Tree Transformations and Caching for Efficient Reduction

9.1 Motivating Example

Even with the use of extended context-free grammars, there remain cases where HDD performs suboptimally. We will use a running example of manageable size throughout this chapter to highlight some issues and help in discussing our proposed improvements. In our example, a fictional program takes a JSON [17] file as input, but has some restrictions on how data has to be organized: it must contain exactly one array of single-digit non-zero integers. However, we assume that someone has made an implementation mistake and instead of strictly checking the input against these restrictions only JSON syntax is validated and that numbers in the array fall between 1 and 9, inclusively. We also assume that this sloppiness, i.e., that floating-point numbers can slip through input validation, will cause a failure in our program. The corresponding testing function — using the notations from Zeller et al. — is given in Figure 9.1a, while a failure-inducing input is shown in Figure 9.1b.

Since we want to use Hierarchical Delta Debugging to find a reduced test case that still reproduces the failure, we will first use a grammar-based parser to build a tree from the input and compute the smallest allowable syntactic fragments for each node. In Figure 9.1c, we give a trimmed-down (but still valid and useful-for-the-purpose) version of the JSON grammar in ANTLR v4 format [74], in Figure 9.1d, we present the calculated minimal replacements, and in Figure 9.1e, we show the parse tree built from our input.

The results of running HDD on our example, both the output tree and the regenerated JSON representation, are shown in Figures 9.1f and 9.1g. Note that the first element of the output array (2) is not necessary to reproduce the failure, but the structure of the grammar (the “value (’,’ value)*” part) and the minimal replacement of token ‘,’ being itself do not allow it to be erased completely. Moreover, it cannot be replaced with \( \Phi[NUM] = '0' \) either as the resulting test case would not be failure-inducing anymore (even the “sloppy” input validation would detect that one of the
test(c) =
\begin{cases} 
\times & \text{if valid.JSONArray}(c) \\
\land \forall e \in c : 1 \leq e \leq 9 \\
\land \exists e \in c : e \neq [e], \\
\checkmark & \text{otherwise.}
\end{cases}

array elements is out of bounds). Thus, although the output is not minimal byte or token-wise, it is 1-tree-minimal [65]. The removal of that component is not in the scope of this thesis.

9.2 Vertical Tree Squeezing

As is visible even from our small example, parse trees can contain linear components: paths where each node has at most one child. In the example, the \textit{json–value–array} subgraph and all the \textit{value–NUM} subgraphs are such components. Intuition tells us that such linear components cause unnecessary test attempts: if HDD already visited the topmost node of a linear component at a given level and DD decided to keep it, then it will not decide otherwise for the rest of the nodes of the subgraph either, as it progresses to the next levels. If the minimal replacement strings of the nodes are identical, then the opposite is true as well, i.e., marking any node in the subgraph as removed will yield the same output. Thus, collapsing such lines into a single node can squeeze the height of the tree and reduce the number of tests.

The optimization, which can be applied to the root of the tree before HDD is actually invoked, is formalized as algorithm \texttt{squeezeTree} in Listing 9.1. The auxiliary predicate \texttt{isToken} holds for nodes representing tokens (or terminals) from the grammar, \texttt{children} gives the number of children of internal nodes, while \texttt{child} selects one of those children.

Figure 9.2 shows the effect of the algorithm on our JSON example. In the tree, only paths of length 2 are collapsible (\textit{json–value} and \textit{value–NUM}, but not \textit{value–array}
Listing 9.1. The Squeeze Tree Algorithm

```
procedure squeezeTree(node)
begin
if not isToken(node) then begin
    forall i in 1..children(node) do
        child(node, i) := squeezeTree(child(node, i))
    if children(node) = 1 and \( \Phi(node) = \Phi(child(node, 1)) \) then
        return child(node, 1)
    end
return node
end
```

![Squeeze Tree Algorithm Diagram](image)

Figure 9.2. The effect of the squeeze tree algorithm: (a) the tree of Figure 9.1e after squeezing and (b) the squeezed tree after HDD.

as \( \Phi[\text{value}] \neq \Phi[\text{array}] \). However, in experiments with other inputs and grammars, we have seen even 18 node long linear paths squeezed (more on that in Section 9.5). Also note that no JSON output is listed because the output is identical to what is shown in Figure 9.1g.

### 9.3 Hiding Unremovable Tokens

Another interesting thing we can notice in our example is that even though some of the nodes are marked as removed in the HDD-reduced tree (those crossed out), they show up in the output nevertheless. The reason behind it is that the smallest allowable syntactic replacement computed for some tokens from the grammar is identical to the occurrence of the token in the input. This is especially true for the anonymous tokens, like \([\text{'}, \text{'}, \text{'}, \text{']}]\) in the example, but it can occur in general, too. The consequence is that no matter how DD decides, whether to keep them (in which case the actual token text will be used) or not (in which case the replacement will contribute to the output), it will yield the same test case. But DD will try to keep and remove them anyway, leading to superfluous test attempts.

However, if we already marked such “unremovable” tokens as removed in a preprocessing step, before applying HDD to the tree, then they would not be collected by tagNodes, effectively hiding them from ddmin. The optimization is formalized as the algorithm hideUnremovableTokens in Listing 9.2. The auxiliary function text gives the textual representation of a token as it appeared in the input, while markAsRemoved is the same as used during HDD, i.e., instead of completely removing a node, it replaces it with the computed syntactic fragment (and prevents it from being tagged ever again). The effect of the algorithm on the parse tree of the example is shown in Figure 9.3. (The tree after HDD is not shown again as it is identical to the tree of Figure 9.1f. But the decrease we expect is not in the size of the output but in the number of test cases.)
Chapter 9. Tree Transformations and Caching for Efficient Reduction

Listing 9.2. The Hide Unremovable Tokens Algorithm

```plaintext
procedure hideUnremovableTokens(node)
begin
if isToken(node) then
  if text(node) = Φ(node) then
    markAsRemoved(node)
  else
    forall i in 1..children(node) do
      hideUnremovableTokens(child(node, i))
end
```

Figure 9.3. The effect of the Hide Unremovable Tokens algorithm: the tree of Figure 9.1e after the algorithm.

9.4 Content Caching

Delta Debugging, the basis of HDD, has configuration of changes as its key concept. The changes may be the lexical decompositions of an input, but in general they can stand for other notions of changes as well. Because of this generality, although DD implementations do apply caching of test outcomes, these outcomes are usually (actually, in all DD implementations known and available to us) associated with configurations. Furthermore, HDD is not only defined based on Delta Debugging and on the configurations of the nodes of a tree level, but is often also implemented with a generic DD variant. However, while configuration-based caching can be good enough for generic Delta Debugging, it may not be the best approach for HDD: various configurations of tree nodes at a given level may induce the same output, configurations of tree nodes on different levels may induce the same output, and configurations of different HDD iterations may also induce the same output – which means that all such configurations will yield the same test outcome, too. Unfortunately, if the caching of outcomes is done on the basis of the tree nodes of a given level, none of these recurrences can be detected, i.e., the outcomes will not be retrieved from the cache but will require repeated test attempts.

In Figure 9.4, we show two examples of node configurations from different tree levels yielding the same outputs (all of them taken from the actual execution of HDD on our running example). In Figures 9.4a and 9.4b, nodes from different levels constitute the configuration, they even determine distinct trees, but still generate the same passing (i.e., invalid but not failure-inducing) output. Similarly, in Figures 9.4d and 9.4e, distinct trees determined by node configurations from two different levels cause the generation of the same failing output.
Figure 9.4. Various configurations tested by HDD during the minimization of the example of Figure 9.1: (a) a passing configuration of the 4th level, (c) a passing configuration of the 5th level, (b) the output of both passing configurations, (d) a failing configuration of the 4th level, (f) a failing configuration of the 5th level, and (e) the output of both failing configurations.

While configuration-based caching cannot prevent the re-testing of such trees, associating outcomes with the content of the output does remove such redundancies. Therefore, we propose to optimize HDD by using content-based caching instead of the traditional configuration-based one.

9.5 Evaluation

To evaluate the aforementioned ideas in practice, we implemented them in the 17.1 releases of the Picire [41] and Picireny [42] projects. Since caching fundamentally belongs to the \textit{ddmin} algorithm, the new content-based approach was placed in \textit{Picire}. On the other hand, trees are HDD-related concepts and, accordingly, the tree preprocessing steps went into \textit{Picireny}.

The experiments were executed on a platform with an Intel Core i7 CPU clocked at 2200 MHz equipped with 16GB RAM. The machine was running macOS Sierra 10.12.2 with kernel Darwin 16.3.0.

For the sake of comparability, our experiments start with re-evaluating test cases from prior works [65, 44] and from the previous chapter. These consist of a Java, a C, and an HTML source, and also a test case with the mixture of HTML, CSS, and JavaScript. The Java source contains a division by zero, the C example discovered a crash in GCC in version 2.95.2, while the HTML-only and the mixed content-type tests caused assertion failures in the rendering engine of Google Chrome (Blink) at revision 402879. Additionally, we used our running JSON example from Section 9.1, and a real
Table 9.1. Baseline

<table>
<thead>
<tr>
<th>Test</th>
<th>Input File &amp; Tree</th>
<th>HDD (1 / *)</th>
<th>Output File &amp; Tree (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Height</td>
<td>Rules</td>
</tr>
<tr>
<td>JSON</td>
<td>33 B</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>Java</td>
<td>18804 B</td>
<td>35</td>
<td>8521</td>
</tr>
<tr>
<td>C</td>
<td>672 B</td>
<td>75</td>
<td>1364</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>26901 B</td>
<td>57</td>
<td>9411</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>880882 B</td>
<td>123</td>
<td>36544</td>
</tr>
</tbody>
</table>

Table 9.2. Effect of Squeeze Tree

<table>
<thead>
<tr>
<th>Test</th>
<th>Optimized Tree</th>
<th>HDD (1 / *)</th>
<th>Output Tree (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Height</td>
<td>Rules</td>
<td>Tests</td>
</tr>
<tr>
<td>JSON</td>
<td>4 (67%)</td>
<td>8 (50%)</td>
<td>22 / 27 (85% / 79%)</td>
</tr>
<tr>
<td>Java</td>
<td>28 (90%)</td>
<td>60 (73%)</td>
<td>60 / 74 (95% / 93%)</td>
</tr>
<tr>
<td>C</td>
<td>32 (43%)</td>
<td>307 (24%)</td>
<td>301 / 355 (47% / 36%)</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>73 (72%)</td>
<td>7101 (69%)</td>
<td>324 / 305 (92% / 87%)</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>107 (87%)</td>
<td>24822 (86%)</td>
<td>15459 / 21995 (91% / 95%)</td>
</tr>
</tbody>
</table>

Table 9.3. Effect of Hide Unremovable Tokens

<table>
<thead>
<tr>
<th>Test</th>
<th>Optimized Tree</th>
<th>HDD (1 / *)</th>
<th>Cache Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tokens</td>
<td>Tests</td>
<td>Cache Hits</td>
</tr>
<tr>
<td>JSON</td>
<td>7 (47%)</td>
<td>28 / 28 (95% / 90%)</td>
<td>17 / 21 (97% / 92%)</td>
</tr>
<tr>
<td>Java</td>
<td>58 (75%)</td>
<td>75 (94%)</td>
<td>92%</td>
</tr>
<tr>
<td>C</td>
<td>514 (91%)</td>
<td>264 (66% / 85%)</td>
<td>78 (28%)</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>343 (97%)</td>
<td>56 (98% / 96%)</td>
<td>2781 (43%)</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>91122 (35%)</td>
<td>49900 (27% / 85%)</td>
<td>11207 / 19467 (77% / 84%)</td>
</tr>
</tbody>
</table>

Table 9.4. Effect of Content-based Cache

<table>
<thead>
<tr>
<th>Test</th>
<th>HDD (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
</tr>
<tr>
<td>JSON</td>
<td>19 (73% / 56%)</td>
</tr>
<tr>
<td>Java</td>
<td>55 (87% / 69%)</td>
</tr>
<tr>
<td>C</td>
<td>298 / 489 (43% / 32%)</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>241 / 322 (68% / 53%)</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>868 / 1181 (76% / 62%)</td>
</tr>
</tbody>
</table>

Table 9.5. Effect of Content-based Cache

<table>
<thead>
<tr>
<th>Test</th>
<th>HDD (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
</tr>
<tr>
<td>JSON</td>
<td>19 (73% / 56%)</td>
</tr>
<tr>
<td>Java</td>
<td>55 (87% / 69%)</td>
</tr>
<tr>
<td>C</td>
<td>298 / 489 (43% / 32%)</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>241 / 322 (68% / 53%)</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>868 / 1181 (76% / 62%)</td>
</tr>
</tbody>
</table>

Table 9.6. Effect of Content-based Cache

<table>
<thead>
<tr>
<th>Test</th>
<th>HDD (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
</tr>
<tr>
<td>JSON</td>
<td>19 (73% / 56%)</td>
</tr>
<tr>
<td>Java</td>
<td>55 (87% / 69%)</td>
</tr>
<tr>
<td>C</td>
<td>298 / 489 (43% / 32%)</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>241 / 322 (68% / 53%)</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>868 / 1181 (76% / 62%)</td>
</tr>
</tbody>
</table>
Chapter 9. Tree Transformations and Caching for Efficient Reduction

Figure 9.5. Relative number of test cases executed by HDD*

Table 9.5. Overall improvement

<table>
<thead>
<tr>
<th>Test</th>
<th>HDD (1 / *)</th>
<th>Tests</th>
<th>Cache Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON</td>
<td>17 / 17 (65% / 50%)</td>
<td>15 / 22 (60% / 76%)</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>53 / 53 (84% / 66%)</td>
<td>1 / 15 (– / –)</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>163 / 281 (25% / 18%)</td>
<td>97 / 372 (25% / 46%)</td>
<td></td>
</tr>
<tr>
<td>HTML</td>
<td>308 / 309 (87% / 68%)</td>
<td>58 / 132 (102% / 203%)</td>
<td></td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>755 / 1037 (66% / 55%)</td>
<td>603 / 1154 (62% / 89%)</td>
<td></td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>11608 / 17253 (68% / 74%)</td>
<td>49600 / 53127 (27% / 28%)</td>
<td></td>
</tr>
</tbody>
</table>

by 12–57%. (The longest squeezed lines were 2 for the JSON test case, 5 for Java, 18 for C, and 3 for all HTML-containing tests. The outstanding results of the C test case are attributed to the highly chained nature of the expression productions in the available C grammar.) However, this decrease of rule nodes is not directly commensurate with the drop in the number of executed test cases: removing nodes is only effective if they are in a kept subtree. On the other hand, removing inner nodes changes the structure of the tree and so the content of the certain levels, which alters the result of DD on that level. In general, we can say that we could decrease the number of needed test cases by at least 5% and with a maximum of 64%.

Table 9.3 shows what happens when the unremovable tokens are hidden from ddmin. This change cannot alter the number of rule nodes; it has an effect on the number of token nodes only and hence the steps and cache hits needed for reducing. As the first column shows, 53–72% of the tokens are completely useless since their unparsed representation is exactly the same both in kept and removed states. However, as was also experienced with tree squeezing, this remarkable reduction of participating nodes does not correlate explicitly with the number of executed test cases, because ddmin behaves differently on the changed configuration. In this case, the gain we achieved in the test runs is 2% minimum and 23% at most.

Table 9.4 shows the outcome of investigating content-based caching. This change does not alter the tree at all, so its influence is confined to the number of run test cases and cache hits. According to the results, content-based caching is a powerful optimization: it caused a 13–68% decrease in the number of test runs thanks to reusing 2–325% more outcomes from cache.

After evaluating the impact of the optimizations separately, we combined them to measure how they worked together. Since our primary goal is to decrease the number of needed test executions, we present this information in Figure 9.5. The shade and pattern of the bars denote the applied optimizations. The first white bars serve as a baseline that marks the unoptimized implementation. If a bar’s pattern contains north-west lines, then the unremovable tokens are hidden. The presence of north-east
lines denotes that tree squeezing was applied. If a bar has a gray background, then we used content-based caching while running HDD*. As the previous tables already revealed, hiding nodes and squeezing the tree decreases the amount of executed test cases, but their efficiency strongly depends on the structure of the tree and the place of the failure-inducing subtree. The bars also show that if we apply only one of the optimizations, then content-based caching is the most powerful. However, if we take a look at the combined results, then we get a bit more nuanced view. When token hiding and tree squeezing are applied together, their effects on the number of spared tests approximately add up. On the other hand, when we pair them with content-based caching, then the gain is not so obvious, if there is any. This behavior arises from the fact that all of the optimizations intend to eliminate duplicate test runs, either by squeezing the input tree, or by hiding the unremovable tokens, or by explicitly caching the executed test cases. However, while tree-squeezing and token-hiding usually eliminate different kinds of recurrences, content-caching easily overlaps with both of them diminishing their efficiency.

In Table 9.5, we summarize the overall gain achieved when applying all optimizations. As the table shows, we could spare at least 13% of the executed tests, but depending on the test case, this ratio could even reach 82%.

9.6 Conclusions

In this chapter, we have presented three optimization ideas to decrease the number of test executions during Hierarchical Delta Debugging, thereby improving its efficiency: the squeeze tree and the hide unremovable tokens algorithms, and the idea of content-based caching. The techniques have been implemented in publicly available projects and evaluated on a set of test cases featuring six different input formats (JSON, Java, C, HTML, CSS, and JavaScript). Separately, they could achieve 64%, 23%, and 68% reduction in the testing steps, respectively, but when combined, this went up to 82% in the best case, equating to a more than 5-fold speedup of the minimization process.
“Testing proves a programmer’s failure. Debugging is the programmer’s vindication.”

— Boris Beizer

10

Coarse Hierarchical Delta Debugging

10.1 Motivating Example

In Figure 10.1a, we show a real failure-inducing input, a small JavaScript program that crashed the JerryScript [54] execution engine created for resource-constrained IoT devices at version 1.0. The rest of the figure presents the application of HDD to this test case. Figure 10.1b shows part of the JavaScript (officially, ECMAScript) grammar (in ANTLR [73] v4 notation), while Figure 10.1c gives the smallest allowable syntactic replacements for the relevant parser and lexer rules using the $\Phi$ notation. Figures 10.1d and 10.1e show the parse trees of the input – built by a parser generated from the grammar –, before and after HDD. Finally, Figure 10.1f presents the minimized failure-inducing script. In the parse trees, the crossed out nodes denote those “pruned off”, i.e., which appear with their minimal replacements in the output. Note that some nodes are marked as pruned or removed even in the input parse tree: these are tokens whose actual text in the input is the same as their smallest allowable syntactic fragment. (E.g., we can always try to replace a string with the empty string literal '"""' and keep the test case valid, at least syntactically, but there is no shorter variant for the 'function' keyword.) As it will make no difference in the output whether these nodes are marked as kept or not, and it reduces the workload of DD if they are marked as removed, it is worth removing them before HDD is applied, as discussed in the previous chapter.

If we take a closer look at the output parse tree in Figure 10.1e, we can notice that in addition to the nodes which were already removed in the input tree, the only nodes marked for removal are the nodes labeled with a * symbol. These nodes group those parts of the input which correspond to quantified expressions in the grammar, e.g., to +-repeated occurrences of sourceElement or to the ?-optional sourceElements expression. (Note that quantifiers – or regular expressions, in general – on the right-hand side of grammar rules require extended context-free grammars, which are proven to describe exactly the context-free languages, just like standard context-free grammars do.) A useful feature of these *-nodes is that they have an empty minimal replacement.
Figure 10.1. An HDD example of a JavaScript program that crashes JerryScript 1.0: (a) the failure-inducing script, (b) an excerpt from the ECMAScript grammar, (c) minimal replacement strings calculated from the grammar, (d) parse tree built from the input using the grammar, (e) the tree after HDD, and (f) the minimized failure-inducing script.

10.2 Coarse Hierarchical Delta Debugging

The observation we made in Figure 10.1 gave us the idea to make hierarchical delta debugging focus on the repeating or optional parts of the parse tree, i.e., on nodes with $\Phi[.] = ',$. The effectively complete removal of such subtrees is expected to bring the
biggest gain in terms of test case reduction, while other parts of the parse tree, which are not allowed by syntax to disappear, deserve less attention. Therefore, we have created a variant of HDD called Coarse Hierarchical Delta Debugging, which visits all levels of the tree one by one like the original HDD, but filters out those nodes from the input configuration of DD that have a non-empty replacement fragment. In the (not that extreme) case when a level has no empty-Φ nodes, the algorithm performs no testing steps related to that level and leaves the tree completely unchanged. The pseudocode of the algorithm is given in Listing 10.1: all auxiliary routines are the same as explained for HDD in Section 6.2, except for filterEmptyPhiNodes, which performs the above described filtering, i.e.,

\[
\text{filterEmptyPhiNodes}(\text{nodes}) = \{ n : n \in \text{nodes} \land \Phi(n) = '' \}.
\]

We do acknowledge that with the presented JavaScript example, we were “lucky”. Its grammar uses quantifiers to express lists and optional parts, and it has empty-Φ nodes in its parse tree, which produces the extreme situation that not only does Coarse HDD give a “good enough” output in fewer steps than HDD, but also exactly the same 1-tree-minimal output as HDD. Of course, this can and will not always be the case – neither for the above defined Coarse HDD, nor for its fixed-point iteration (which we will denote as Coarse HDD*, analogously to HDD* being the iterated version of HDD).

### 10.3 Tree Flattening for Improved Coarse HDD

As a second example, we present a C program that crashed the GCC 2.95.2 compiler. It was first shown in the paper of Hildebrandt and Zeller [36] and repeated in a slightly modified, but still crash-inducing form by Misherghi and Su [65]. Here, in Listing 10.2, we list that second variant.

Figures 10.2a and 10.2b reveal that even HDD* cannot produce a result that we would call minimal (note the stray semicolons), although it fits the definition of 1-tree-minimality, while Coarse HDD* leaves even more extras in the output. The cause for the superfluous parts in both outputs is that the C grammar we used for building the input parse tree formalized various list constructs with the help of left-recursive rules instead of quantifiers, as exemplified in Figure 10.2c. Since the grammar uses quantifiers very sparsely, significant parts of the input tree have non-empty smallest allowable syntactic fragments (see Figure 10.2d). For HDD, this prevents the deletion
Listing 10.2. C program that crashes GCC 2.95.2.

```c
1 double mult(double z[], int n)
2 {
3     int i;
4     int j;
5     for (j = 0; j < n; j++) {
6         i = i + j + 1;
7         z[i] = z[i] * (z[0] + 0);
8     }
9     return z[n];
10 }
11
12 int copy(double to[], double from[], int count)
13 {
14     int n = (count + 7) / 8;
15     switch (count % 8) {
16         case 0: *to++ = *from++;
17         case 7: *to++ = *from++;
18         case 6: *to++ = *from++;
19         case 5: *to++ = *from++;
20         case 4: *to++ = *from++;
21         case 3: *to++ = *from++;
22         case 2: *to++ = *from++;
23         case 1: *to++ = *from++;
24     } while (--n > 0);
25     return (int)mult(to, 2);
26 }
27
28 int main(int argc, char *argv[])
29 {
30     double x[20], y[20];
31     double *px = x;
32     while (px < x + 20)
33         *px++ = (px - x) * (20 + 1.0);
34     return copy(y, x, 20);
35 }
```

of unnecessary statements and function definitions and allows their replacement with semicolons only, while for Coarse HDD, this prevents any processing of the affected nodes.

Of course, one way of solving the problem would be to manually rewrite the grammar to use quantified constructs extensively. However, that would require a one-time manual effort at least, and potentially even more maintenance work later if the grammar with its original structure would have to be kept for non-test-reduction purposes. Thus, we aimed at dealing with the issue in an automated way.

We can apply a tree transformation to the input that looks for left and right-recursive tree constructs that encode lists (an example is shown in Figure 10.2e). If we find a matching pattern, we can flatten all “list elements”, i.e., move them below newly created *-nodes at the same tree level. Such a transformation yields trees as if lists were not parsed by rules of $L : \alpha | L \beta$ format (where $\alpha$ and $\beta$ stand for arbitrary compositions.
Double $a(\text{double } z[], \text{int } a)$
{
  int $i$;
  int $j$;
  for (;;) {
    $i = i + 1$;
    $z[i] = z[i] * (z[0] + 0)$;
  }
}
Listing 10.3. The Flatten Tree Recursion Algorithm

```c
procedure FLATTEN_TREE_RECURSION(node)
    forall child in CHILDREN(node) do
        FLATTEN_TREE_RECURSION(child)
    end forall
    num ← |CHILDREN(node)|
    if num > 1 then
        if NAME(CHILDREN(node)[1]) = NAME(node) then
            left ← CHILDREN(node)[1]
            right ← CHILDREN(node)[2..num]
            CHILDREN(node) ← CHILDREN(left) + EMPTY_NODE(right)
        elseif NAME(CHILDREN(node)[num]) = NAME(node) then
            left ← CHILDREN(node)[1..num-1]
            right ← CHILDREN(node)[num]
            CHILDREN(node) ← EMPTY_NODE(left) + CHILDREN(right)
        end if
    end if
end procedure
```

double a(double z[])
{
    int i;
    int j;
    for (; ;) {
        j = i + j + 1;
        z[i] = z[i] * (z[0] + 0);
    }
}

double mult(double z[])
{
    int i;
    int j;
    for (; ;) {
        j = i + j + 1;
        z[i] = z[i] * (z[0] + 0);
    }
}

Figure 10.3. The effect of tree recursion flattening on HDD* and Coarse HDD*:
(a) the minimized output produced by HDD*, (b) the minimized output produced
by Coarse HDD*, and (c) part of the transformed parse tree of the program.

As a final remark, we have to note that there are cases, of course, where even
tree recursion flattening (or a manual grammar rewrite) will not help Coarse HDD
to approximate HDD as closely as shown above. Then, we can experiment with concate-
nating the two test case reduction algorithms making Coarse HDD* a preprocessing
step to HDD*, thus relying on the minimality guarantees of HDD* but benefiting from
the smaller input tree (pruned by Coarse HDD*) that needs to be worked with. The
evaluation of this idea, together with both algorithms presented earlier in this section,
is detailed in Section 10.4.
### 10.4 Evaluation

To evaluate Coarse Hierarchical Delta Debugging, we have created Python 3-based prototype implementations of the algorithms shown in the previous section and published them in the 17.7 release of the Picireny [42] project. The experiments were executed on a platform with Intel Core i7 CPU clocked at 2200 MHz equipped with 16GB RAM. The machine was running macOS Sierra 10.12.2 with kernel Darwin 16.3.0.

For the sake of comparability, our experiments start with re-evaluating test cases from prior works [65, 44, 48] that were also discussed in the previous chapters. These consist of a Java, a C, and an HTML source, and also a test case with the mixture of HTML, CSS, and JavaScript. Additionally, we used a JSON input on an artificial tester, a real life HTML and JavaScript source from GitHub’s website that caused an assertion failure in the rendering engine of Apple Safari (WebKit) at revision 199605, and the JavaScript program that crashed JerryScript 1.0 (shown in Section 10.1). All the tests were processed with grammars from the official ANTLR repository [1].

First, we show the baseline details of these test cases in Table 10.1. The first group of data gives size metrics of the input (i.e., size in bytes, the numbers of different nodes in the input tree, and the height of the tree). The second group describes the number of testing steps performed by the HDD and HDD* algorithms on the input tree, as well as the iteration count of HDD*. The final group shows the output metrics of the result (listing two numbers separated with a slash where the outputs of HDD and HDD* differ).

The following three tables (10.2, 10.3, and 10.4) show how Coarse Hierarchical Delta Debugging, the Flatten Tree Recursion-preprocessed HDD, and the likewise preprocessed Coarse HDD perform compared to our baseline. From Coarse HDD*, we expected to lower the number of executed tests significantly while yielding somewhat larger outputs. As Table 10.2 shows, the drop in test count ranges between 42% and 79%, which we consider promising. Of course, we also have to consider the output size increases before drawing a conclusion: in the same table, we can find that the growth ranges from +0% to +182%, compared to the original output sizes. The almost 3 times bigger output for the HTML & JS test case can be somewhat frightening at first sight, however, if we take a closer look, we can notice that this only means that now the reduced output is not 0.06% of the original input but “only” 0.18%.

Our expectations towards the Flatten Tree Recursion algorithm were to have a positive effect on the output size while allowing some slight increase in the testing steps – but, if applied before Coarse HDD*, still keep the test count lower than for HDD*. In our experiment, the flattening algorithm did produce some unexpected results if applied before HDD* (as shown in Table 10.3, the size of the reduced HTML & CSS & JS test case increased by 9%), but if it worked together with Coarse HDD*,

<table>
<thead>
<tr>
<th>Test</th>
<th>Input File &amp; Tree</th>
<th>Output File &amp; Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Height</td>
</tr>
<tr>
<td>JSON</td>
<td>33 B</td>
<td>4</td>
</tr>
<tr>
<td>JS</td>
<td>74 B</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>740 B</td>
<td>31</td>
</tr>
<tr>
<td>Java</td>
<td>18804 B</td>
<td>27</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>29610 B</td>
<td>50</td>
</tr>
<tr>
<td>HTML</td>
<td>31173 B</td>
<td>73</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>880882 B</td>
<td>105</td>
</tr>
</tbody>
</table>
testing steps were always fewer than for HDD* (compare Tables 10.1 and 10.4) and maximum 10% more than for Coarse HDD* alone, while output sizes were always smaller than or equal to the results of Coarse HDD* (compare Tables 10.2 and 10.4). The results of the C test case exemplify especially well what we expected from the flattening algorithm (the drop in testing steps remained 79% but the output size did not only decrease compared to Coarse HDD* alone but also compared to HDD*). So, altogether, the results are: 32–79% fewer testing steps at the cost of maximum 139% bigger outputs (but we have to mention again that this increase means +0.09% only with respect to the input size), with an outlier where the output size has not increased but decreased by 13%. On the average of the 7 test cases, this means 58% fewer tests for 31% bigger results, or alternatively, for a growth less than 0.36% of the input size.

In the last table, we show the results of our experiment where we have applied Tree Recursion Flattening first, then Coarse HDD* as a middle step, and the original HDD as the final phase. We hoped that such a pipeline would allow us to reach the same output size as HDD* but faster. As Table 10.5 shows, the results of this experiment are not conclusive. While four test cases out of seven meet the expectations, three others show unexpected behaviour: one needs more steps to reach the output of HDD* (Java, +6% tests), another yields bigger output than HDD* (HTML & JS, +115% size), and

---

Table 10.2. Coarse HDD

<table>
<thead>
<tr>
<th>Test</th>
<th>Coarse HDD (1 / *)</th>
<th>Output File &amp; Tree (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Iter</td>
</tr>
<tr>
<td>JSON</td>
<td>6 (35%)</td>
<td>2</td>
</tr>
<tr>
<td>JS</td>
<td>8 / 9 (42% / 41%)</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>51 / 72 (27% / 21%)</td>
<td>2</td>
</tr>
<tr>
<td>Java</td>
<td>31 (58%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>315 / 378 (42% / 37%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML</td>
<td>152 (49%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>3731 / 4674 (32% / 27%)</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 10.3. HDD with Flatten Tree Recursion

<table>
<thead>
<tr>
<th>Test</th>
<th>HDD (1 / *)</th>
<th>Output File &amp; Tree (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Iter</td>
</tr>
<tr>
<td>JSON</td>
<td>17 (100%)</td>
<td>2</td>
</tr>
<tr>
<td>JS</td>
<td>19 / 22 (100%)</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>223 / 386 (119% / 113%)</td>
<td>3</td>
</tr>
<tr>
<td>Java</td>
<td>54 (102%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>776 / 1060 (105% / 103%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML</td>
<td>308 / 309 (100% / 100%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>12755 / 18537 (110% / 108%)</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 10.4. Coarse HDD with Flatten Tree Recursion

<table>
<thead>
<tr>
<th>Test</th>
<th>Coarse HDD (1 / *)</th>
<th>Output File &amp; Tree (1 / *)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Iter</td>
</tr>
<tr>
<td>JSON</td>
<td>6 (35%)</td>
<td>2</td>
</tr>
<tr>
<td>JS</td>
<td>8 / 9 (42% / 41%)</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>45 / 73 (24% / 21%)</td>
<td>3</td>
</tr>
<tr>
<td>Java</td>
<td>35 / 36 (66% / 68%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>376 / 464 (51% / 45%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML</td>
<td>152 (49%)</td>
<td>2</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>4420 / 6269 (38% / 37%)</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 10.5. Flatten Tree Recursion + Coarse HDD* + HDD

<table>
<thead>
<tr>
<th>Test</th>
<th>Coarse HDD* + HDD (1/*)</th>
<th>Output File &amp; Tree (1/*)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Iter</td>
</tr>
<tr>
<td>JSON</td>
<td>6 + 4 (59%)</td>
<td>2 + 1</td>
</tr>
<tr>
<td>JS</td>
<td>9 + 8 (89% / 77%)</td>
<td>2 + 1</td>
</tr>
<tr>
<td>C</td>
<td>73 + 167 / 169 (128% / 71%)</td>
<td>3 + 2</td>
</tr>
<tr>
<td>Java</td>
<td>36 + 20 (106%)</td>
<td>2 + 2</td>
</tr>
<tr>
<td>HTML &amp; CSS &amp; JS</td>
<td>464 + 474 / 664 (126% / 110%)</td>
<td>2 + 2</td>
</tr>
<tr>
<td>HTML</td>
<td>152 + 90 / 91 (79% / 79%)</td>
<td>2 + 2</td>
</tr>
<tr>
<td>HTML &amp; JS</td>
<td>6269 + 2649 / 4150 (77% / 61%)</td>
<td>4 + 3</td>
</tr>
</tbody>
</table>

Figure 10.4. Results of the reduction algorithm variants relative to HDD*: (a) number of executed test steps and (b) size of the reduced test case.

yet another exhibits both problems (HTML & CSS & JS, +10% tests, +9% size). Note that theory is not failing, the bigger outputs are still 1-tree-minimal: depending on the order of the node removals, we can reach different minimal trees that cannot be pruned further one-node-at-a-time, and they generate different outputs. Unfortunately, this observation just explains the symptom, but does not help reduce the tests further in practice.

To summarize the results of Tables 10.1–10.5, we chart the two key metrics in Figure 10.4: the number of executed testing steps performed by all (fixed-point iterated) algorithm variants on all test cases, normalized to the results of HDD*, and the size of the outputs, likewise. In the charts, northwest-hatched bars denote that the HDD* algorithm was used, northeast hatching signals the use of Coarse HDD*, while a gray background denotes that the input parse tree was preprocessed by the Tree Recursion Flattening algorithm. Any combination of these patterns means the appropriate combination of the algorithms, e.g., the right-most gray crosshatched bars stand for the results of Table 10.5, i.e., all algorithms pipelined.

In addition to the comparison of the end results (i.e., how big an output we achieved and how fast), we were also interested in the speed of convergence to the result, especially in the size of the intermediate result of HDD* after exactly as many steps as Coarse HDD* took. To visualize this, we have plotted the intermediate test case sizes
for all reduction algorithm variants as a function of testing steps in Figure 10.5. In the graphs, the color of the line denotes whether the Tree Recursion Flattening algorithm was applied (gray) or not (black). The solid lines represent the progress of the original Hierarchical Delta Debugging algorithm, the dashed lines stand for Coarse HDD, and the dash-dotted lines show how HDD continues work after Coarse HDD* finishes. The empty and full circles mark the end of the first and last iterations of an algorithm, respectively. (Note that for some test cases, some graph features “disappear”, e.g., no black lines or empty circles are visible. This signals that the behaviour of some algorithms is not affected by tree recursion flattening, or that the iterated version of an algorithm performs no more tests after the end of the first iteration.)

We can see from the graphs that Coarse HDD on the flattened tree always approximates the result significantly faster (the dashed gray line is always way below the solid black line), and when Coarse HDD* finishes (when we reach the first full circle on the
dashed gray line), the intermediate result of HDD* (solid black line) is usually still much larger, sometimes by orders of magnitude, e.g., HTML & JS in Figure 10.5g. The same almost holds for Coarse HDD without tree recursion flattening, too, but the C example with the highly recursive grammar from Section 10.2 works again as a worst case example (see Figure 10.5c).

Throughout this section, we have used the number of testing steps as the key metric in measuring the effect of the introduced algorithms on the performance of test case reduction. To put these numbers in a practical context, we have measured the elapsed “wall-clock” time during the reduction of the biggest test case (the real-life HTML & JS input) with some of the algorithm variants and combinations: it took 5 hours and 39 minutes for the baseline HDD* to produce the minimal test case, while Coarse HDD* has finished in 1 hour and 53 minutes. This saves 67% from the execution time, which is actually more than three and three-quarter hours. Even if Tree Recursion Flattening is applied before Coarse HDD*, which increases the number of required testing steps, the execution time is still 3 hours or 54% shorter than for HDD* (2 hours and 36 minutes). Finally, the time saving of the pipelined variant (with Tree Recursion Flattening and Coarse HDD* before HDD*) is still hours faster than the baseline (runs for 3 hours and 30 minutes).

### 10.5 Conclusions

In this chapter, we have introduced the Coarse Hierarchical Delta Debugging algorithm and its fixed-point iteration, Coarse HDD*, variants of HDD and HDD*, that simplify test cases, but without guaranteeing theoretical minimality. In exchange for the potentially bigger results, they are expected to give their results in fewer steps than their original counterparts. Early enough, we have identified that Coarse HDD/HDD* could not perform well on trees that result from test cases parsed by grammars using left or right-recursive rules for list constructs, therefore, we have also proposed a tree transformation as a preprocessing step to mitigate the problem. To see what we really gain – and what we lose – we have evaluated an implementation of the algorithms on various test cases. Some were artificial examples, but others came as real bugs from open-source projects. Altogether, they covered three programming languages (C, Java, and JavaScript) and three data formats (CSS, HTML, and JSON). The two key algorithms presented in this chapter, Coarse HDD* and Flatten Tree Recursion have achieved their goals together: they have reduced test cases with 58% fewer steps, on average, with a maximum gain of 79%. For our largest test case, this has meant several hours faster test case reduction. The price of the speed-up was an average increase in the output sizes by 31%. The maximum observed size increase of 139% may seem considerable, but the increase of the output size is usually negligible when we compare it to the size of the original input: then the increase for the above mentioned worst test case becomes a tiny 0.09%, and never gets higher than 0.36% for any of the test cases.
Part III

Practical Applications
11 Fuzzinator

11.1 Motivation

Although the idea of random or fuzz testing is not a new one [63], the scientific research of the topic is still very active. Multiple papers are published by various authors and research groups every year on how fuzzing can be applied to various targets – e.g., to web engines [30], operating system internals [11], compilers [98], interpreters [50], user interfaces [2], etc. – or how random test case generation can be made more efficient – e.g., using constraint solvers [25], coverage information [6], evolutionary approaches [34, 80], or machine learning [26]. However, fuzzing attracts the attention of not only the members of the scientific community, but also of the industry, and even of unaffiliated enthusiasts: a wide variety of fuzzers is available on the Internet at different levels of maturity [52].

However, as soon as the complexity of the tested application grows beyond that of a “Hello World!” program, random test case generators – no matter how cleverly written – are not enough on their own. Undoubtedly, there is no fuzzing without a test generator (see Part I), but to be useful in practice, it needs to be accompanied by some similarly important components. First of all, whenever a new test case is generated, it must be transported to the system-under-test: even simple command line tools may have different needs, as some expect input directly from the console while others require the test case contents to be written into a file and the file name to be specified on the command line; and more complex applications – e.g., those with graphical user interfaces, or communicating over the network – may demand the setup of a proportionally complex transport infrastructure. Then, as the strength of fuzzing is that it automatically generates large amounts of test cases, it is also important to automatically recognize when a test case drives the SUT into unexpected behaviour. This can be a program crash, a non-zero exit code, an uncaught exception message on the error channel, or a timeout of a remote service, etc. Moreover, not only is it important to detect errors in the SUT, but it is also useful to recognize unique bugs, as random test generators can – and tend to – make SUTs fail similarly (e.g., access illegal memory from the same machine instruction, throw an exception at the same
location, fail on the same assertion) on different inputs. Keeping multiple test cases that trigger the same error is not only superfluous, but it can also lead to the waste of resources when it comes to the analysis of the found bugs. Finally, as fuzzed test cases are randomly generated, even if they trigger a failure in the SUT, they rarely contain the minimum information that is needed for the reproduction of the error, but rather they are full of useless content and only a sub-1% is interesting. Thus, automatic test case reducers can come handy, as it was shown in Part II.

The above highlighted features are almost a must for any practical fuzzing setup, but depending on the use-case, some extras can be considered as ‘nice-to-have’: rapidly evolving SUTs may benefit from an automated update mechanism that fetches, builds, and inserts the latest version of the SUT into the fuzzing infrastructure to ensure that computing power is not wasted on outdated code; or it can improve project management if the found – and hopefully minimized – failure-inducing test cases are automatically reported to an issue tracking system, thus reducing the time the project team needs to react to it.

Unfortunately, many academic, industrial, or independent projects either still focus only on the test case generator component and give no support for the rest of the tasks, thus making their solution hard to put in practice, or they tightly couple several components into a monolithic artifact which cannot be easily reused by or adapted to the needs of others. (Or they claim to have all the necessary advantages, but they are not freely and openly available.)

In this chapter, we introduce an open-source random testing framework, called Fuzzinator \cite{46, 38}, that aims at connecting all the important components of a fuzzing setup in a highly configurable modular way thus helping the re-use and the easy integration of independently developed projects. Thus, Fuzzinator removes the need for reinventing the wheel of fuzzing over and over again.

During the design of Fuzzinator, we have had two sets of requirements: one for what tasks we wanted it to automate, and another for how we wanted it achieved. The requirements on ‘what’ has already been discussed in the paragraphs above, while the ‘how’ was broken down to three sub-requirements: we wanted that it should help the integration of existing tools by requiring no programming from the user in basic use-cases, that it should enable arbitrary customization and extension if existing solutions are not satisfactory, and that it should advocate and facilitate the re-use of its building blocks (be them built-ins or extensions).

Section 11.2 gives details on the high-level architecture of Fuzzinator (the ‘what’), Section 11.3 discusses the way its finer-grained components work together (the ‘how’), and finally, Section 11.4 gives a working example to show how all pieces fit together.

### 11.2 Architecture

Put simply, Fuzzinator is nothing but a job scheduler. But it is a scheduler that deals with jobs tailored for the tasks of fuzz testing. First and foremost, it has the concept of fuzz jobs, which take a fuzzer to generate a batch of test cases and call a SUT with the test cases one-by-one. Whenever the SUT runs into an issue that was not seen before, it records it in a database and schedules a reduce job for running. The task of these jobs is to connect reducers to SUTs, i.e., to make a test case reducer generate simpler versions of a previously found failure-inducing input and feed these versions into the same SUT that was called by the fuzzer to find whether they still reproduce the issue.
Once a reduce job finishes, its result is used to update the database with the compact version of the test case. Moreover, when a reduce job drives its SUT into a previously unseen issue (which is not that frequent, but does happen, as the reduction process is usually also stress-testing the SUT), that is also recorded and a separate reduce job is scheduled again. The scheduler also works with the concept of update jobs, which have the duty to detect when a SUT becomes outdated (e.g., based on timestamps or VCS information) and then perform a SUT-specific update. Finally, as SUTs can change over time, the scheduler can run validation jobs to check whether a previously found and recorded issue can still be reproduced on the current SUT version; and when an issue becomes obsolete it is marked so in the database. Similarly to reduce jobs, if the corresponding test case triggers another new issue in the SUT during validation, it will go though the record and reduce process again.

Figure 11.1 (especially its bottom part) gives a high-level view of the above mentioned concepts.

11.3 Components

The previous section only enumerated the job concepts of the core scheduler of Fuzzinator, but was intentionally vague about what ‘generating a batch of test cases’, ‘calling a SUT’, or ‘detecting an issue’ meant. These concepts belong to a lower level of abstraction and require the mentioning of some implementation details. Fuzzinator is a Python 3-based framework using the NoSQL MongoDB system as its database. Within the framework, every fuzzer is simply represented by a function (or a callable, to use a more generic Python term), and whatever data such a fuzzer function returns is considered as a new test case. Similarly, a SUT is also just a function that shall take
Chapter 11. Fuzzinator

the result of the fuzzer function and return something that it considers as an adequate description of an issue (if it failed, nothing otherwise), the only requirement being that it be a dictionary (a key-to-value mapping) so that the framework can add annotations later when necessary (e.g., to put the test case that triggered the issue in the mapping). Reducers follow the same logic, i.e., they are functions that take a SUT (the function) and a test case (the data previously returned by the fuzzer function) and return something that is a more compact version of its input, but keeps the SUT return a similar issue description. And finally, the identification of unique issues is also working on the above abstractions by enabling the SUTs to put a special key-value pair in the returned dictionary, which is then checked for equivalence in comparisons (to prevent comparing the whole contents of issue dictionaries).

This – seemingly simple – generic representation of all actors (i.e., they are all Python functions) and artifacts (i.e., they are dictionaries of arbitrary structures) opens up the possibility of great modular flexibility and extensibility. If the SUT is a project written in Python and exposes an API, its functions can be directly called and if assertion errors are raised, they can be wrapped into a dictionary to signal the found failure. If the SUT is a command line tool that reads from the console, we can write a simple wrapper that uses Python’s standard libraries to start a child process, pipe the test case to the standard input, and create a new dictionary with the exit code if it was not zero. Similarly, a fuzzer can be a simple Python function constructing arrays of arbitrary bytes, or a command line tool that generates files in some directory, wrapped into a small Python callable to execute the tool and return either the contents or the file system paths of the generated files (depending on the needs of the SUT). Moreover, as functions can wrap or decorate other functions in Python, it becomes easy to separate orthogonal aspects into separate building blocks. E.g., various fuzzer implementations may generate random web content, but all those fuzzers should not implement their own web server to deliver the test cases – the fuzzer functions should just focus on the content generation and let a decorator be applied to handle the transport over HTTP, if it is so expected by the SUT.

To make Fuzzinator useful out-of-the-box, we have analyzed the fuzzing scenarios we had faced in the past and identified their common aspects and their differences. We have turned the smallest units of common aspects into parametric building blocks (i.e., SUT call, fuzzer, reducer, updater functions, etc.) and made them part of the framework. Table 11.1 gives a list of the more important built-in building blocks thus created. Hopefully, the most common fuzzing setups will not require additional programming, but can re-use these built-in components.

The glue between these components is a standard INI configuration file format with predefined section and property names. The configuration files can name Python entities (functions) and their arguments, as well as their order of composition – and the dynamic inspection capabilities of Python enable and ensure their proper execution. The next section gives a concrete example how this works in practice.

11.4 Example

In this section, we show how a working configuration file can be created for the fuzz testing of a real target. We use the JerryScript project [54] as an example, which provides a Linux command line wrapper around the engine for testing purposes.

We start with a minimum configuration that defines one SUT and one test generator
using built-in functions, as shown in Listing 11.1. The sections of the configuration starting with a `sut.` prefix define how SUTs will be handled and also give a name to the SUT (the rest of the section name after the prefix). In the example, the `call` property of the `sut.jerry` section defines that `StdinSubprocessCall` (which writes the test case to the stdin of its subprocess target and returns an issue dictionary if the target exits with a non-zero code) will be used for executing the `jerry` SUT. Parameters of the call (those specific to `StdinSubprocessCall`) are defined in the `sut.jerry.call` section: how (`command`) and where (`cwd`) the target must be executed. Sections starting with `fuzz` connect SUTs with test case generators. In the example, the `fuzz` section references the appropriate SUT section and the `RandomContent` fuzzer that simply produces random strings.

As registering all non-zero exits of the SUT as issues is not really useful in practice,
we should filter out all issues when the SUT did not exit with specifically interesting exit codes and did not print messages to stderr matching a given pattern. We can make use of `StdinSubprocessCall` as it puts exit code information and stdout, stderr dumps into its issue dictionaries, and of built-in decorator solutions that can use this data, as shown in Listing 11.2 (giving only the incremental changes compared to the minimal configuration). The `call.decorate(N)` properties declare the decorators that can access the input and output of the wrapped call, and can modify them. Their parameters are defined in parameter sections named `sut.jerry.call.decorate(N)`.

Issues can not only be filtered but also extended with arbitrary information that helps describing the circumstances of the failure, which can also be implemented with decorators. Listing 11.3 shows how platform, git version, and ID information can be added using built-in solutions: `PlatformInfoDecorator` adds an extra `platform` field to the issue dictionary, filled with OS information, `SubprocessPropertyDecorator` adds a user-defined field (version) with the output of a user-defined script, and `UniqueIdDecorator` combines existing fields (`msg` and `path` fields previously found by `RegexFilter`) into an ID to help detect whether an issue is unique or a duplicate of an already known one.

Finally, we can also control how our SUT is updated and how the test cases of the
Listing 11.4. Updating the SUT and reducing tests.

```plaintext
[sut.jerry]
update_condition=fuzzinator.update.TimestampUpdateCondition
update=fuzzinator.update.SubprocessUpdate
reduce=fuzzinator.reduce.Picire

[sut.jerry.update_condition]
age=12:00:00
path=${sut.jerry.call:cwd}/build/bin/jerry

[sut.jerry.update]
command=git pull origin master &&
   ./tools/build.py --debug --clean
cwd=${sut.jerry.call:cwd}
```

Figure 11.2. Screenshot of Fuzzinator.

found issues should be reduced. Listing 11.4 shows the use of built-in building blocks again: `TimestampUpdateCondition` triggers the update based on the last modification time of the target binary, `SubprocessUpdate` updates JerryScript using a script, and `Picire` reduces test cases with the help of the Picire [41] project.

The here presented four short listings show how multiple aspects of the fuzzing of a real target can be customized with a simple configuration file and using built-in components only. However, the framework is open to replace or mix any and all parts of the setup with custom-coded external building blocks, if necessary.

Figure 11.2 shows a screenshot of Fuzzinator configured for multiple SUTs – including the above introduced JerryScript – in action. (The shown text-based user interface is one of the two options currently available, the other is a classic line-based logger.)
11.5 Practical Results

In this part, we have introduced Fuzzinator, an open-source fuzzer framework. When designing Fuzzinator, our primary goals were to create a framework that helps the integration of existing tools by requiring *no programming* from the user in basic use-cases, enables arbitrary *customization and extension* if existing solutions are not satisfactory, and advocates and facilitates the *re-use* of its building blocks (be them built-ins or extensions).

This framework was used during the evaluation of the test generator introduced in Part I and it is still actively used with other fuzzers, too. Two of these often used fuzzers are Grammarinator [49, 40] and Generinator: RATS [39] that are also our works and that are open-source projects, too. Grammarinator is a grammar-independent fuzzer, which interprets ANTLRv4 parser grammars and uses them as a generator model. This generator found more than 100 issues by itself in the aforementioned JerryScript engine. On the other hand, Generinator: RATS analyses existing HyperText Markup Language (HTML) and Cascading Style Sheet (CSS) sources, and based on the extracted information, it generates new test cases for web browsers. In addition to our own fuzzer projects, we have used other state-of-the-art test generators with Fuzzinator as well. E.g., Fuzzinator even has built-in support for American Fuzzy Lop (AFL), which is the most popular among them. These fuzzers discovered the browser issues that were used for evaluating the reducers in Part II, which are also natively supported by the framework.

We have been using the introduced technology for years now, while hundreds of issues were found, minimized and reported. We have generated inputs in formats already discussed (JavaScript, HTML, CSS) as well as some that are not covered in this thesis (like Scalable Vector Graphics and Mathematical Markup Language). With these inputs, we have tested multiple JavaScript engines, platforms, and rendering engines from both desktop and the resource-constrained domains. Among our targets were the JerryScript, Escargot, and Duktape lightweight JavaScript engines, the IoT.js service platform, and the engines under the hood of several modern web browsers (WebKit and JavaScriptCore from Apple Safari, Blink and v8 from Google Chrome, and ChakraCore from Microsoft Edge). Moreover, we have also tested some loosely related projects using these input formats, e.g., we have also fuzzed the Inkscape vector graphics editor. Table 11.2 summarizes the more than 1000 unique issues that we discovered and reported until the closing of this manuscript.

<table>
<thead>
<tr>
<th>Unique issues</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WebKit (jsc included)</td>
<td>391</td>
</tr>
<tr>
<td>Blink (v8 included)</td>
<td>333</td>
</tr>
<tr>
<td>JerryScript</td>
<td>238</td>
</tr>
<tr>
<td>IoT.js</td>
<td>40</td>
</tr>
<tr>
<td>Duktape</td>
<td>19</td>
</tr>
<tr>
<td>Escargot</td>
<td>15</td>
</tr>
<tr>
<td>ChakraCore</td>
<td>9</td>
</tr>
<tr>
<td>Inkscape</td>
<td>4</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>1049</td>
</tr>
</tbody>
</table>
Appendix
A.1 Summary in English

In this thesis, two areas of software testing are presented, namely the random testing of JavaScript engines and the automatic reduction of interesting or failure triggering test cases. According to the two topics above, the structure of this thesis is split into two main parts and an additional third one containing practical results. Below, the summary of each part is presented.

In Part I, we introduced a new JavaScript API fuzzing approach. Fuzzing techniques can be sorted into two categories: mutation or generation-based solutions. Mutational approaches require an initial seed test suite that will be mutated by applying various transformations to them. On the other hand, generational approaches use – mostly manually built – generator models and create test cases according to these models. In this thesis, a new hybrid solution is introduced, which has properties both from generative and mutational approaches. Our approach is generational in the sense that it generates test cases from scratch, based on a model named the Prototype Graph, and it is mutational since the model is – partially – built from information extracted from existing test cases. The model extraction happens in two steps: first, in discovery step, we load a specially crafted JavaScript source to the examined engine, which creates our initial model of the target engines type API. In an optional next step, we improve our model with information extracted from existing test cases using the aforementioned introspection capabilities again. To describe our model, we have defined a graph-based formalization of type information in JavaScript. We have shown how to automate the building of such graphs to describe the API exposed by a JavaScript execution engine, and we have also defined an API fuzzing method based on graph terms. To the best of our knowledge, this is the first work to use graph formalization and traversal for representing type information of JavaScript and use it for fuzzing. In addition to the formal definitions, we have also presented the experimental results of a prototype implementation of a JavaScript engine API fuzzer. The results show that the prototype graph-based API fuzzing technique can be on par with available solutions in terms of total code coverage and even yield significantly better coverage results in JavaScript.
API-related modules. With respect to its issue finding capabilities: the prototype tool triggered real program crashes in JavaScriptCore during the experiments. Since the original publication of the presented research, the implementation of the technique has been actively used and it found more than 100 unique issues in the JerryScript, Duktape, and IoT.js JavaScript engines and platforms, which were reported to and fixed by the developers of the tested projects.

Part II was dedicated to the area of automatic test case reduction. We evaluated two state-of-the-art automatic test case reduction approaches, pinpointed their weaknesses and proposed improvements to them.

First, we have taken a close look at the minimizing Delta Debugging algorithm, which is the most widespread solution for syntax unaware automatic test case reduction tasks. Our original motivation was to enable the algorithm to utilize the parallelization capabilities of multi-core and multi-processor computers, but as we analyzed it, we found further enhancement possibilities. We recognised the parallelization potential implicitly given in the existency checks of the original definition of the algorithm (although never exploited by its authors), but we have also pointed out that the same original definition unnecessarily prescribed sequentiality elsewhere, for its “reduce to subset” and “reduce to complement” steps. Moreover, we have argued that the “reduce to subset” step of the original algorithm may even be omitted completely without losing its key property of resulting 1-minimal test cases while being more effective for some types of inputs. Each observation has led to an improved algorithm variant, all of which are given in pseudo-code. Finally, we presented an experiment conducted on 4 artificial test cases and on 2 wide-spread browser engines with 3 real test cases each. All test cases were minimized with all presented algorithm variants and with 12 different levels of parallelization (ranging from single-core execution – i.e., no parallelization – to 64-fold parallelization on a computer with 48 virtual cores). The results of the 1098 successfully executed test case minimizations prove that all presented improvements to the Delta Debugging algorithm achieved performance improvements, with the best variants reducing the running time on real test cases significantly, by cca. 75–80%.

Second, we turned to the syntax-aware Hierarchical Delta Debugging algorithm and its implementation, and proposed improvements to address their shortcomings. We have found that although HDD itself is not strictly dependent on how the tree it is working on is built, both its discussion and its reference implementation are tied to standard context-free grammars. We have argued that using extended context-free grammars for HDD is beneficial in several ways, as the use of quantifiers in the right-hand side of grammar rules instead of recursive rules yields more balanced parse trees and enables an improved strategy for “smallest allowed syntactic fragment replacement”, which in turn can result in a smaller output. In addition to the grammar-related proposals, we have pointed out practical issues that hinder the use of the reference implementation. All the enhancement ideas and usability improvements are embodied in a modernized HDD implementation. Experimental evaluation of the created tool supports the grammar-related ideas outlined in this part: its reduced outputs are significantly smaller (by cca. 25–40%) on the investigated test cases than those produced by the original HDD implementation using standard context-free grammars.

As further optimizations, we have presented three ideas to decrease the number of test executions during Hierarchical Delta Debugging, thereby improving its efficiency: the “squeeze tree” and the “hide unremovable tokens” algorithms, and the idea of “content-based caching”. These ideas were added to our aforementioned prototype
According to our evaluation, these optimizations together caused more than 5-fold speedup of the minimization process in the best case.

Finally, we have introduced the Coarse Hierarchical Delta Debugging algorithm and its fixed-point iteration, Coarse HDD*, variants of HDD and HDD*, that simplify test cases, but without guaranteeing theoretical minimality. In exchange for the potentially bigger results, they are expected to give their results in fewer steps than their original counterparts. Additionally, we have also proposed a tree flattening as a preprocessing step to improve the performance of Coarse HDD/HDD* on trees that result from test cases parsed by grammars using left or right-recursive rules for list constructs. We have evaluated an implementation of the algorithms on various test cases: some were artificial examples, but others came as real bugs from open-source projects. Altogether, they covered three programming languages (C, Java, and JavaScript) and three data formats (CSS, HTML, and JSON). The two key algorithms presented in this thesis, Coarse HDD* and Flatten Tree Recursion have achieved their goals together: they have reduced test cases with 58% fewer steps, on average, with a maximum gain of 79%. For our largest test case, this has meant hours faster test case reduction. The price of the speed-up was an average increase in the output sizes by 31%. The maximum observed size increase of 139% may seem considerable, but the increase of the output size is usually negligible when we compare it to the size of the original input: then the increase for the above mentioned worst test case becomes a tiny 0.09%, and never gets higher than 0.36% for any of the test cases.

In Part III, we introduced Fuzzinator, an open-source modular fuzzer framework that – among many other features – integrates the aforementioned JavaScript generator and test case reduction approaches and which assisted us in finding, reducing and reporting more than 1000 unique issues in various open-source projects.

A.2 Main Results of the Thesis and Contributions of the Author

In this thesis, five main results were stated. As the thesis consists of two main parts, the results are also separated into two parts. In the list below, the author’s contributions to these results are clearly stated.

Fuzz Testing of JavaScript Engines

The main result of the first part of the thesis is the introduction of a novel representation model of JavaScript type APIs and the definition of its usage for random testing arbitrary JavaScript engines. The result, which is based on publication [43], is presented in Chapter 4.

1. Prototype Graph-based Fuzzing of JavaScript Engines

The author, jointly with her co-author, defined and formalised a graph-based model, named the Prototype Graph, to represent the type system of arbitrary JavaScript engines. The model is able to describe the object-based type system of the JavaScript language including the expected signature of the methods and constructors provided by the underlying JavaScript engine. The author invented how the introduced model
can be automatically extracted from any JavaScript engine and how it can be extended by analysing existing test cases.

With the help of this model, the author designed and formalized an algorithm to generate random JavaScript expressions from it and stress-test the underlying JavaScript engine. She implemented a prototype tool which is able to build engine-specific models, to extend them with information extracted from existing test cases, and to use the model for fuzz testing real life JavaScript engines. Based on the prototype tool, she evaluated and compared it to another state-of-the-art JavaScript fuzzer and reported the found issues.

Automatic Test Case Reduction

In the second part, two state-of-the-art automatic test case reduction approaches are analysed, their weaknesses are pinpointed and several improvements are proposed to them. The results, which are based on publications [45, 44, 48, 47], are presented in Chapters 7, 8, 9, 10 respectively.

2. Decomposition of the Delta Debugging Algorithm

The author has analysed the widespread Delta Debugging algorithm, with special focus on its ddmin variant. She recognised, that the two components of ddmin, i.e., the subset and complement-based reduction parts, are not necessary sequential: they can be reordered and one of them is even avoidable while still ensuring the original 1-minimality of the result. Additionally, she has also shown that the ordering of the test executions is irrelevant from the perspective of 1-minimality. This observation has led the author to define a parallel version of the ddmin algorithm. The author, jointly with her co-author, created a clean room implementation of ddmin which incorporates the aforementioned findings. Using this prototype tool, the author performed an extensive evaluation to empirically analyse the effectiveness of the approach on artificial and real-life examples.

3. Analysis of the Effects of Different Parser Grammars on Hierarchical Delta Debugging

The author investigated the syntax-aware version of Delta Debugging, named Hierarchical Delta Debugging, which applies ddmin to the levels of parse trees. She recognised, that using trees built by standard context-free grammars is not optimal since they can be heavily unbalanced due to recursive rules. She proposed to use extended context-free grammars to build the input trees and designed an improved algorithm for the minimal replacement string calculation. The author and her co-author created a prototype implementation and used it to compare the result of Hierarchical Delta Debugging on standard and extended context-free grammar-based trees. Using this tool, the author performed the experimental evaluation of the proposed idea.
4. Improvements to the Hierarchical Delta Debugging with Tree Transformations

The author has recognised that some constructs in the parse tree trigger superfluous test executions. She identified two such constructs and suggested to apply two preprocessing algorithms to the trees before the actual reduction: “tree squeezing” and the “hiding of unremovable tokens”. The author, jointly with her co-authors, implemented these algorithms to the aforementioned prototype tool and evaluated their effects.

5. Coarse Hierarchical Delta Debugging

The author and her co-authors have observed that not all nodes of parse trees contribute equally to the process of reduction. They recognised that the most gain comes from the removal of subtrees that can be actually deleted from the output (i.e., syntax does not require some replacement string). Based on this observation, the author created the coarse version of Hierarchical Delta Debugging, which only considers such nodes of parse trees during reduction and she designed the algorithm of tree flattening. She implemented this variant into the aforementioned prototype tool and evaluated its effect on real life reduction tasks.

A.3 Magyar nyelvű összefoglaló

A disszertáció a szoftvertesztelés két területével foglalkozik, nevezetesen a JavaScript motorok véletlenszerű vagy fuzz tesztelésével és a hibát kiváltó tesztesetek automatikus redukciójával. Ezen két témakörnek megfelelően a tézis is két fő része lett osztva, valamint tartalmaz egy harmadik részt is a gyakorlati eredményekről. Alább ezen részek főbb eredményeit foglaljuk össze.

Az I. részben bemutatunk egy új tesztgenerálási megközelítést, amelyet JavaScript API-k fuzz tesztelésére használnak. A véletlenszerű tesztelési technikákat jellemzően két kategóriába szokás sorolni: mutációs és generatív megoldásokra. A mutációs megoldások létező tesztekből állítanak elő újakat, különböző transzformációk alkalmazásával. Ugyanakkor a generatív megoldások teljesen új teszteket állítanak elő valamilyen generátor modell alapján, melyet többnyire manuálisan készítenek el. Ebben a tézisben egy olyan hibrid megoldást mutatunk be, amely mind a generatív, mind pedig a mutációs megközelítésekre jellemző tulajdonságokat is tartalmaz. A megoldásunk generatív abban az értelemben, hogy a teszteket modell alapján generáljuk, amit Prototípus Gráfnak nevezünk, ugyanakkor a modell létező tesztekből kinyert információkat is tartalmazhat, amely viszont a mutációs megközelítések sajátja. A modell elkészítése két lépésben történik: először egy feltérképező lépésben a tesztelt motorba betöltünk egy speciális szkriptet, amely a nyelv önelemző képességét használva elkészít egy alap modellt. Ezután, egy opcionális második lépésben, létező tesztekből kinyert információval pontosítjuk a modellünket ismételten az önelemző képesség kihasználásával. A modellünk leírására egy új, gráf alapú formalizációt definiáltunk, amely képes ábrázolni a JavaScript nyelv típusrendszerét. Megmutattuk hogyan nyerhetünk ki ilyen gráfot automatikusan egy tetszőleges JavaScript motorból, amely leírja annak API-ját, illetve hogy a kinyert gráf miként használható a motor fuzz tesztelésére. Legjobb tudásunk szerint ez az első olyan kutatás, amely gráf alapú modelllel ábrázolja JavaScript futtató motorok API-ját és annak bejárásával állít elő teszteseteket.
formális definíció mellett módszerünket a gyakorlatba is átültettük, és egy prototípus implementációs segítségével megvizsgáltuk annak gyakorlati használhatóságát. Az eredmények azt mutatták, hogy az automatikusan előállított modellből generált tesztek a jelenlegi legmodernebb, manuálisan előállított modellből generáló modellel hasonló eredményt ér el a teljes kód lefedettség tekintetében. Ellenben ha csak a JavaScript típus API-ját leíró modulokat tekintjük, akkor az általunk javasolt megoldás lényegesen jobban teljesíti a lefedettséget illetően, mint a kézzel írt modellből generáló megoldás. A módszer hibafelderítési képességét tekintve a prototípus implementációjának már az első kísérletek során is több valós hibát talált a JavaScriptCore motorban. A kutatás során előállt eszközt azóta is aktívan használjuk és alkalmazzuk más JavaScript motorokra is, például a JerryScript, a Duktape és az IoT.js JavaScript motorokra és platformra, melyekben több mint 100, korábban ismeretlen, egyedi hibát találtunk és reportáltunk, melyek azóta javítva is lettek az egyes projektek fejlesztői által.

A II. rész a teszeset-redukálás témakörének szenteltük. Kielemzettünk két korszerű automatikus tesztredukoló módszert, meghatározottuk a hiányosságait és megoldásokat javasoltuk azok javítására. Először a minimalizáló Delta Debugging algoritmust vettük görögén, ami jelenleg az egyik legelterjedtebb automatikus szintaxis-független redukoló megközelítés. Noha az eredeti célunk csak az algoritmus párhuzamosítása volt, hogy jobban kihasználhassuk a modern, több magos számítógépek kapacitását, azonban az analízis során kiderült, hogy további potenciális javítások is szóba jöhetnek. Felismertük, hogy a párhuzamosítás lehetőségét már az algoritmus eredeti definíciója is magában foglalta, noha a szerzők ezt soha nem használták ki. Emellett arra is rájötöttünk, hogy az eredeti algoritmus szükségentelelő definíálja szekvenciáisának a “minimalizálás részhalmazra” és a “minimalizálás complementerre” lépéseket. Továbbá azt is bebizonyítottuk, hogy a komplett “rézshalmazra való minimalizálás” lépés elhagyható anélkül, hogy az eredmény kulcsfontosságú tulajdonságának, az 1-minimalitásnak a garanciáját elveszítenénk. Minden fentebb említett felismerés egy-egy javított algoritmus variánshoz vezetett, melyeket pseudokódokkal meg is adtunk. Végül átfogó kísérleti kiértékelést végeztünk, amihez 4 mesterséges és 3 valós életből vett minimalizáló teszten használtunk fel. Minden tesztet leredukáltunk az összes bemutatott algoritmus-variánszal, 12 különböző párhuzamosítási szinten (a párhuzamosítás mértéke az 1 processzes – párhuzamosítás nélküli – verziótól, a 64-szeres párhuzamosítási szintig terjedt). Az 1098 minimalizálás eredménye bebizonyította, hogy minden bemutatott javítás a minimalizáló Delta Debugging algoritmusán teljesítményjavulást eredményezett, melynek mértéke a legjobb esetben 75–80%-nyi futásidő csökkentés volt. Ezután a szintaxis-követő Hierarchikus Delta Debugging algoritmust és annak prototípus implementációját vizsgáltuk meg, majd az alapján javítási lehetőségeket vázoltunk. Felismerítettük, hogy noha az algoritmus tetszőleges fákra alkalmasító, minden eredeti értékezés, mind pedig a referencia implementáció hagyományos környezetfüggetlen nyelvtanok kal korlátosan felépített fákkal dolgozik. A tézisben megmutattuk, hogy a kiterjesztett környezetfüggetlen nyelvtanok használata több szempontból is előnyösebb. Ezek támogatják a mennyiségjelzők használatát a nyelvtani szabályok jobb oldalán, amellyel kiválthatóak a hagyományos környezetfüggetlen nyelvtanok rekurzív konstrukciói, ezáltal kiegynézoljottabb elemzési fákat eredményezve. Továbbá egy kifinomultabb stratégiát tesznek lehetővé a “legkisebb szintaktikus részkifejezés” meghatározására, amely végül kisebb kimeneti fát és teszeszetek eredményez. A nyelvtannal kapcsolatos javaslataink mellett gyakorlati problémákat is lokalizáltuk a referencia implementációban, amelyek megnehezítették annak használthatóságát. Az általunk készített
modernizált HDD implementáció tartalmazza az összes elméleti és gyakorlati szempontból megfogalmazott ötletünket és továbbfejlesztési javaslatunkat. A kísérleti kiértékelés megmutatta, hogy a modernizált implementáció lényegesen (hozzávetőleg 25–40%-kal) kisebb eredményt produkált, mint az eredeti, hagyományos környezetfüggetlen nyeltvannal készített fákkal dolgozó megoldás.

További optimalizálásként, három javaslatot tettünk a HDD alapú redukálás során szükséges tesztfuttatások számának csökkentése érdekében: a “fák összenyomását”, a “töröltetlen tokenek elrejtését” valamint a “tartalom alapú gyorsítótárazást”. Ezen ötletek is meg lettek valósítva a korábban említett modernizált HDD implementációkban. Méréseink szerint ezek az optimalizációk a legjobb esetben együttesen több mint 5-szörös gyorsulást eredményeztek a minimalizáló folyamatban.

Végül létrehoztunk egy Coarse (azaz durva felbontású) Hierarchikus Delta Debugging névre keresztelt algoritmus variánsát és ennek fix-pont iterációs változatát, a Coarse HDD*-ot, melyek anélkül minimalizálják a teszteket, hogy garanciát vállalnának az 1-minimalizátrasára. A potenciálisan nagyobb eredményért cserébe azt vártuk, hogy kevesebb lépésben véget érnek majd, mint az eredeti megfelelőjük. A módosított algoritmus mellett javasoltunk még egy “fa kilapító” előfeldolgozó lépést, a Coarse HDD/HDD* sebességének javítását célozva. Ez a transzformáció olyan fák esetében tud javulást elérni, melyek nyelvtana rekurzív szabályokkal ismeri fel a listaszerű konstrukciókat. Az új algoritmusból prototípus implementáció készült, amit különböző mesterséges és valós teszeszetek minimalizálásával értékeltek ki. Az eredmények azt mutatták, hogy a tézisben prezentált két algoritmus, azaz a Coarse HDD* és a “fa kilapító transzformáció” együttesen elérték a kitűnőbb célt: átlagosan 58%-kal kevesebb lépésben hoztak eredményt, míg a legjobb esetben ez az érték 79% volt. A legnagyobb tesztünknél ez a javulás átlagosan 31%-nyi kimenet növekedés volt. A legrosszabb esetben ez 139%-os növekedést jelentett, azonban ez a szám csak a kiemelte méretének arányát tükrözi: amennyiben a kiemelte méretvégtelen méretekhez arányosítjuk, akkor az átlagos növekmény csupán 0.09%-ot tesz ki és sohasem több mint 0.36%.

Végül, a III. részben bemutattuk a Fuzzinator nevű nyílt forrású fuzzer keretrendszerét, amely – sok egyéb funkcionalitás mellett – integrálja a korábbi részekben bemutattott JavaScript generáló és teszeszet redukáló megközelítéseket és amely a segítségünkre volt több mint 1000 egyedi hiba felfedezésében különböző nyílt forrású projektekben.
A.4 Tézispontok és a szerző hozzájárulása az eredményekhez

A disszertáció 5 tézispontot fogalmaz meg. A disszertáció két fő részének megfelelően az eredmények is két részre oszthatók. Alábba a szerző saját hozzájárulásait vesszük számba.

JavaScript motorok fuzz tesztelése


1. JavaScript motorok Prototípus Gráf alapú fuzz tesztelése

A szerző, a társ szerzőjével közösen, definiálta és formalizálta a Prototípus Gráf nevű modellt, amellyel tetszőleges JavaScript motor típus API-ját képes leírni. A modell a JavaScript nyelv objektum-alapú típusrendszerére mellett azt is tartalmazza, hogy a motor által biztosított beépített függvények és metódusok milyen szignatúrákat várnak el. A szerző megmutatta hogyan nyerhető ki a fent említett gráf automatikusan egy tetszőleges JavaScript motorból, illetve hogyan finomítható az létező tesztek elemzésével.

A szerző kidolgozott egy algoritmust, amely a fenti modellt használva képes véletlenszerű JavaScript kifejezéseket generálni és azokkal stressz-tesztelni a JavaScript futtató motort. A szerző továbbá implementált egy prototípus eszközt, amely képes felépíteni az alap modellt, képes azt létező tesztek elemzésével pontosítani, és végül valós motorok fuzz tesztelésére felhasználni. A prototípus eszköz felhasználásával összehasonlította a megközelítést egy másik, korszerű fuzzyerrel, illetve reportálta a talált hibákat.

Automatikus teszteset-redukció

A második részben két modern automatikus teszteset-redukaló megközelítést analizáltunk, tártuk fel a gyenge pontjaikat és tettünk javaslatokat azok továbbfejlesztésére. Ezen eredményeket, melyek a [45, 44, 48, 47] publikációkon alapszanak, a 7, 8, 9, 10 fejezetekben tárgyaltuk részleteiben.

2. A Delta Debugging algoritmus elemeire bontása

A szerző analizálta az elterjedt Delta Debugging algoritmust, különös tekintettel a $ddmin$ variánsára. Felismerte, hogy a $ddmin$ komponensei, a részhalmoz és komponenter alapú redukciós lépések, nem szükségszerűen szekvenciálisak: sorrendjük felcserélhető és egyikük el is hagyható anélkül, hogy elveszítenénk az eredeti algoritmus 1-minimalitásra vonatkozó garanciáját. Továbbá a szerző megmutatta, hogy az egyes tesztek futási sorrendje szintén irreleváns az 1-minimalitás szempontjából.

A szerző, közös eredményként a társ szerzőjével, elkészítette a $ddmin$ új implementációját, amely magában fogalja az imént felsorolt fejlesztéseket. A prototípus implementáció felhasználásával, a szerző széleskörű kiértékelést végzett mesterséges és
valós életbeli tesztek minimalizálásával, hogy a gyakorlatban is igazolja a megközelítés hatékonyságát.

3. Nyelvtanok hatásának elemzése a Hierarchikus Delta Debugging algoritmusra

A szerző megvizsgálta a Delta Debugging szintaxis-követő verzióját, nevezetesen a Hierarchikus Delta Debuggingot, amely a `ddmin` algoritmust alkalmazza a elemzési-fák szintjére. Felismerte, hogy a hagyományos környezetfüggetlen nyelvtanok által épített elemzési fák használata nem optimális ebben a kontextusban, mivel erősen elfajulóak lehetnek a rekurzív szabályok használatának következtében. Ezért javasolta a kiterjesztett környezetfüggetlen nyelvtanok által készített elemzési fákat alkalmazni bemenetként. A szerző, közös eredményként a társszerzőjével, elkészítette a vázolt javaslatok megfelelő prototípus implementációját és ennek segítségével összehasonlították a Hierarchikus Delta Debugging algoritmus hatásfokát hagyományos és kiterjesztett környezetfüggetlen nyelvtanokkal épített fákon futtatva.

4. A Hierarchikus Delta Debugging javítása fatranszformációkkal

A szerző felismerte, hogy néhány konstrukció a elemzési-fában felesleges tesztfuttatásokat eredményez. Két ilyen konkrét konstrukciót azonosított, amelyekhez két algoritmust dolgozott ki – a “fa összenyomást” illetve a “törölközetlen token elrejtését” –, amelyeket előfeldolgozó lépésként alkalmazott a bemeneti fára. A szerző, közös eredményként a társzerzőivel, implementálta a transzformációkat a fentebb említett prototípus eszköz részeként és kiértékelte annak hatásait.

5. Coarse Hierarchikus Delta Debugging

A szerző, a társzerzőivel közösen, felismerte, hogy a elemzési-fák nem minden pontja járul hozzá egyformán a redukcióhoz. Közösen felismerték, hogy a legtöbb nyereség azon részfák eltávolításából adódik, amelyek ténylegesen töröltek a kimenetből, azaz amelyek minimális helyettesítő kifejezése az üres sztring. Ezen felismerésen alapulva, a szerző elkészítette a “Coarse” (durva felbontású) verzióját a Hierarchikus Delta Debugging algoritmusnak, amely csak a fenti tulajdonsággal rendelkező fapontokat veszi figyelembe a redukció során. A szerző implementálta ezen új variánst a korábban említett prototípus eszközbe és kiértékelte annak hatásfokát.
Bibliography


Bibliography


