Modernizing Hierarchical Delta Debugging

Renáta Hodován
University of Szeged
Department of Software Engineering
Dugonics tér 13, 6720 Szeged, Hungary
hodovan@inf.u-szeged.hu

Ákos Kiss
University of Szeged
Department of Software Engineering
Dugonics tér 13, 6720 Szeged, Hungary
akiss@inf.u-szeged.hu

ABSTRACT

Programmers tasked with the fixing of a bug prefer working on a minimal test case where every single bit is needed to reproduce the failure. However, cutting off the excess parts of a potentially large test case can be a tedious and time-consuming task if performed manually, which has led to the research and development of automated test case reduction techniques. The decade-old Hierarchical Delta Debugging (HDD) algorithm targets structured test inputs, parses them with the help of grammars and applies the minimizing Delta Debugging algorithm to the built trees.

We have investigated this algorithm and its implementation, and propose improvements in this paper to address the found shortcomings. We argue that using extended context-free grammars with HDD is beneficial in several ways and the experimental evaluation of our modernized HDD implementation, called Picireny, supports the outlined ideas: the reduced outputs are significantly smaller (by circa 25–40%) on the investigated test cases than those produced by the reference HDD implementation using standard context-free grammars. These results, together with the technical improvements that ease the use of the modernized tool, can hopefully help spreading the adaptation of HDD in practice.

CCS Concepts

•Software and its engineering → Software testing and debugging; •Theory of computation → *Grammars and context-free languages;*

Keywords

Hierarchical Delta Debugging; Extended Context-free Grammars; Parallel

1. INTRODUCTION

Random test generation, or fuzzing [14], is an increasingly popular technique, which has become an integral part of the stability and security testing of various software projects [8]. However, fuzzer-generated failure-inducing test cases tend to be overly large and contain parts that contribute nothing to the failure. (The same may hold for non-generated test cases as well, of course.) However, programmers tasked with the fixing of a reported bug prefer working on a minimal test case where every single bit is needed to reproduce the failure. Cutting off the excess parts of a test case can be a tedious and time-consuming task if performed manually, which has led to the research and development of automated test case reduction techniques.

Perhaps the most well-known and wide-spread automatic technique is the minimizing Delta Debugging (DD) algorithm of Zeller and Hildebrandt [16, 4, 17]. The algorithm is theoretically sound and produces so-called 1-minimal test cases on a selected granularity. However, the selection of the granularity can cause usability issues in practice. In a typical use case where DD is applied to a structured text file, the granularity is either at line or character level. Unfortunately, character units cause unacceptably long running times very often while lines can contain complex structures and thus may result in non-minimal test cases (which are 1-minimal at line granularity, but not minimal from the viewpoint of the bugfixing programmer). To overcome this difference in concepts, Misherghi and Su [9, 10] proposed Hierarchical Delta Debugging (HDD), which makes use of the structure of the test case by parsing it with the help of an appropriate grammar and building an abstract syntax tree (AST). The minimizing Delta Debugging algorithm is then applied to the levels of the AST, thus ensuring that the boundaries of the units the DD is working with do align with the structure of the file.

Even though the idea of HDD is promising, we didn't see it spread in practice. We have investigated related papers and the reference implementation and found several issues – some purely technical, some more intricate – that might have hindered its usage. These findings have led us to modernize HDD, and the result of that work is presented in this paper.

First, in Section 2, we briefly overview Hierarchical Delta Debugging, both ideas behind it and the reference implementation, and then in Section 3, we describe how to deal with the identified issues and introduce *Picireny*, the modernized HDD implementation. In Section 4, we evaluate our approach on three test cases. In Section 5, we discuss related work. Finally, in Section 6, we give a summary of our work and conclude the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

A-TEST'16, November 18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4401-2/16/11...$15.00
http://dx.doi.org/10.1145/2994291.2994296

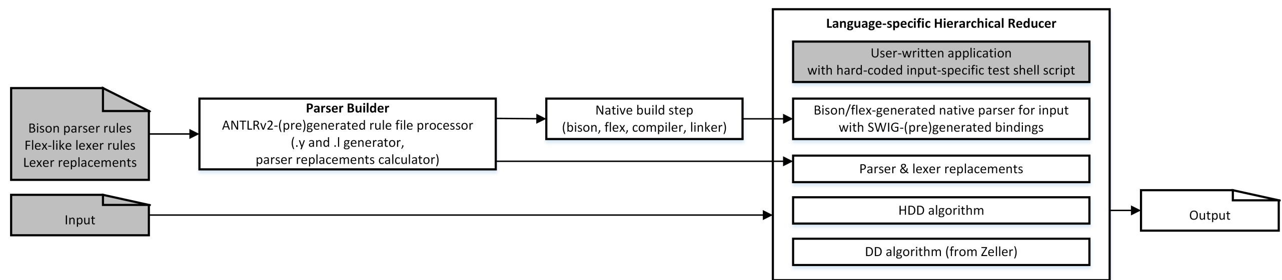

Figure 1: Architecture overview of HDD (user-provided elements marked with grey background).

Listing 1: The Hierarchical Delta Debugging Algorithm

```
1  procedure HDD(input_tree)
2      level ← 0
3      nodes ← TAGNODES(input_tree, level)
4      while nodes ≠ ∅ do
5          minconfig ← DDMIN(nodes)
6          PRUNE(input_tree, level, minconfig)
7          level ← level + 1
8          nodes ← TAGNODES(input_tree, level)
9      end while
10 end procedure
```

2. HIERARCHICAL DELTA DEBUGGING

In their first paper [9], Misherghi and Su define HDD as an iterative algorithm on trees representing hierarchical inputs (parse trees). The HDD algorithm applies minimizing Delta Debugging to each level of a tree starting from the top and progressing downwards. For the sake of completeness, we give the verbatim copy of the pseudocode formulation of HDD in Listing 1.

In a subsequent work [10], Misherghi realizes that pruning a tree during and after DD (i.e., handling subtrees rooted at nodes that are not kept in a configuration by DD) may be more complex than simply omitting the corresponding parts from the input. In order to ensure the syntactic validity of a test case, non-kept nodes (together with their subtrees) should be replaced by the smallest allowed syntactic fragments. For context-free grammars, a fix-point algorithm is given to compute minimal length strings for non-terminals. These minimal strings can act as replacements of inner tree nodes built by a corresponding parser.

Even if the HDD algorithm 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 context-free grammars. Unfortunately, as even the original authors note, 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[1] written mostly in Python 2 and utilizing Zeller's DD routine[2], 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 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., one 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 composed of regular expressions and 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.

[1] http://www.gmw6.com/src/hdd.tgz
[2] https://www.st.cs.uni-saarland.de/dd/DD.py

3. PICIRENY, 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 don't 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 leafs 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

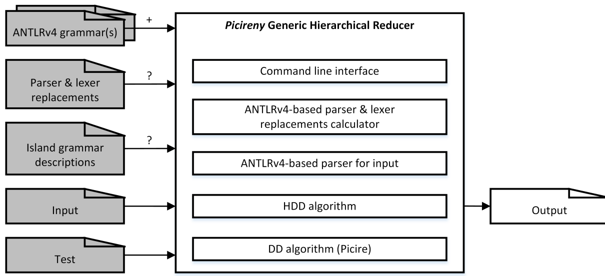

Figure 2: Architecture overview of Picireny.

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 fix-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 in 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*[3], we have implemented the above outlined algorithmic changes, making use of Picire [5], an improved implementation of the DD algorithm, and ANTLRv4 [11], 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 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 4, 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.

[3]https://github.com/renatahodovan/picireny

Listing 2: Java test case reduced by HDD

```
1 final class a   { ; a a ( ) { ; a = 0 / 0 ;
      ; ; ; ; ; ; } ; ; ; ; ; ; }
```

Listing 3: Java test case reduced by Picireny

```
1  class a
2 {
3  void a ()
4 { a =a / 0;
5 }
6 }
```

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.

4. EXPERIMENTAL RESULTS

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, 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[4] also contain a grammar for Java, the

[4]https://github.com/antlr/grammars-v4

Listing 4: Bison parser rule for Java block

```
1  block
2  :  block_begin block_end
3  |  block_begin block_statements block_end
4  ;
5
6  block_statements
7  :  block_statement
8  |  block_statements block_statement
9  ;
```

Listing 5: ANTLR parser rule for Java block

```
1  block
2  :  '{' block_statements* '}'
3  ;
4
5  block_statements
6  :  local_variable_declaration_statement
7  |  statement
8  |  type_declaration
9  ;
```

Table 1: Java example

		HDD	Picireny
Input source	Lines	489	
	Bytes	18804	
	Non-ws chars	13269	
Output source	Lines	1	6
	Bytes	98	37
	Non-ws chars	39	23
Input tree	Height	252	35
	Terminal nodes	5390	5937
	Non-terminal nodes	36802	8821
Output tree	Height	44	21
	Terminal nodes	1	1
	Non-terminal nodes	43	20
	Non-empty non-kept nodes	27	14
Executed tests		82	50
Time (s)		15	9

test case could be executed with both tools. Listing 2 contains the reduced test case produced by HDD, while Listing 3 is the output of Picireny.

Although the results are similar, we can still observe some differences. First, Listing 2 contains everything in a single line, while Listing 3 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 unparse 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 4 and 5 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 1. As the *Output source* section and the output listings show, there is not much difference between them. However, 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 so-called *content_shell* component, which is a minimal browser built on top of the rendering engine of Chrome.

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

Table 2: HTML example

		HDD	Picireny
Input source	Lines	662	
	Bytes	31173	
	Non-ws chars	27660	
Output source	Lines	1	1
	Bytes	478	238
	Non-ws chars	319	238
Input tree	Height	2934	101
	Terminal nodes	3806	1123
	Non-terminal nodes	3638	1359
Output tree	Height	70	96
	Terminal nodes	11	3
	Non-terminal nodes	86	97
	Non-empty non-kept nodes	59	214
Executed tests		203	263
Time (s)		1872	2631

Listing 6: Minimized multi-language test case

```
1  <head><a>
2  <script>window.onload = function() {
3
4  document.execCommand ('selectAll')
5  document.designMode = 'on'
6  document.execCommand ('indent')
7  } </script><style>
8  *, metadata:first-letter {
9  will-change:a
10 }
11 * {;
12 position: fixed
13 } </style>
14 </head><html ><a><body ><a><a><a><a ><a
       ><a><metadata >&shy;&#Xbcf1bdcBf31A;A1Lc
       </a></a></a></a></a></a></a></body></html>
```

Table 3: HTML & CSS & JS example

		Picireny
Input source	Lines	756
	Bytes	29610
	Non-ws chars	26817
Input tree	Height	48
	Terminal nodes	5665
	Non-terminal nodes	8185
Output source	Lines	14
	Bytes	359
	Non-ws chars	327
Output tree	Height	33
	Terminal nodes	52
	Non-terminal nodes	144
	Non-empty non-kept nodes	70
Executed tests		1014
Time (s)		3603

refactoring of an HTML lex 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 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 the Picire DD implementation). 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 wouldn't 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 couldn't compare Picireny to any other tool in this case, we feel it is fair to use parallelism and other improvements. Beside running 4 workers, we have configured the underlying DD algorithm according to the suggestions in Hodován and Kiss [5], and so we ran only complement tests. The minimized test case is shown in Listings 6 and the metrics of the reduction are presented in Table 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.

5. RELATED WORK

The majority of automated debugging techniques can be sorted into two categories. The first approach, just like this work, focuses on the failure-inducing input and tries to find its smallest relevant subset that still produces the expected failure. The other approach is a code-centered solution that tries to reduce the amount of relevant statements contributing to the failure.

Probably the best known technique of the first category is delta debugging [16, 4, 17], which iteratively splits up the input into smaller pieces with increasing granularity and greedily chooses from them until it finds a 1-minimal subset that still produces the expected failing behavior. In spite of its simplicity, it became a widespread method for debugging that motivated many researchers to improve it further. Its main drawback is that sometimes hundreds or even thousands of test cases need to be executed until the final 1-minimal test case is found, which is highly time-consuming. A plausible way of lowering the number of test executions is concentrating on those cases only that fulfill various levels of syntactical requirements. The first related work was *delta*[5] that processed inputs with C/C++-like structures using the *topformflat* tool and put syntactically related tokens on one line before applying the DD algorithm to the input with line granularity. To generalize this approach, Misherghi and Su [9, 10] suggested to parse the inputs with appropriate language grammars and perform splitting along the boundaries of matching rules. This resulted in an order of magnitude less test cases than the original approach. Zhiqiang and Xiangyu have developed a technique [7] to reverse engineer input syntactic structures from executions and used this structure as the input of HDD, eliminating the need for a grammar of the input.

Another input-centered solution, that also operates with grammars is SIMP [1]. Contrary to HDD, SIMP uses the language grammars not only for parsing, but also for analyzing the built AST. As a result of this analysis, SIMP can identify syntactically interchangeable subtrees and it has a strategy to efficiently replace them with each other without breaking the syntactical correctness. Although its primarily goal is to improve the reduction of failing database test cases, the idea can be adapted to other languages too.

C-Reduce [12] is a modular reducer developed for C/C++ sources. During reduction, it iteratively performs source-to-source alternations on the test. The transformations implement such operations that a skilled developer would do while manually executing the minimization, e.g., replacing constants, removing balanced parentheses, removing unused variables, etc. As a result, the authors report reaching 25 times speed-up in the reduction of C/C++ sources over other public tools.

Beside semantic improvements, there was room for technical trade-offs too. One example is the Lithium project[6], which is a clean-room and simplified implementation of the original DD algorithm. Another similar tool is Picire [5], which beside leveraging the technical progress of hardware in the last decade and parallelizing the minimization process of the original DD algorithm, also revealed such tweaks in DD that could significantly improve its efficiency without changing the 1-minimality of the results.

[5] http://delta.tigris.org/

[6] http://www.squarefree.com/lithium/

The solutions falling in the code-centered category try to identify those statements of the failing application that are responsible for the failure. These approaches are usually based on program code analysis, slicing being the most widespread technique among them, either by using its static [15] or dynamic version [6, 18, 3]. Renieris and Reiss [13] use program spectra for fault localization. They expect to have many correct runs of a program beside the one failing instance. During reduction, faulty and correct versions are ran in pairs while monitoring and comparing the executed program entities and identifying the likely faulty statements.

Penumbra [2], which uses dynamic tainting to find the failure inducing input, combines the ideas of the two categories reviewed above. It tracks the flow of inputs along data and control dependencies at runtime and uses this information to identify the failure-relevant parts of inputs after the failure has happened.

6. CONCLUSIONS

In this paper, 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 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 spreading the use of HDD in practice.

Experimental evaluation of Picireny supports the grammar-related ideas outlined in this paper: 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).

For future work, we plan to investigate how the ideas discussed by Bruno [1], i.e., the identification and movement of

syntactically interchangeable sub-trees could be adapted to Hierarchical Delta Debugging. An adapted approach could allow the removal of unnecessary parental structures around failure-inducing subtrees. E.g., in Listing 6, the anchor tags (`<a>` and `</a>`) around the `metadata` tag, which are basically the minimized versions of an unnecessary tag structure, could have been completely removed.

Furthermore, the current HDD algorithm applies DD to levels of the tree, i.e., it cuts across different branches. Although this approach doesn't injure any syntactical requirements, it can break semantic dependencies. To avoid such breaks, we could explore and make use of such semantic dependencies in the input. However, it is not only challenging but would result in losing the algorithm's language independence. Instead, we could heuristically change the order of the tree traversals to decrease the number of test cases containing semantic errors. Picireny (and the underlying DD implementation, Picire) has the ability to run DD in different directions already, so this is a potential course for future research as well.

7. REFERENCES

[1] N. Bruno. Minimizing database repros using language grammars. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*, pages 382–393. ACM, Mar. 2010.

[2] J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*, pages 249–260. ACM, July 2009.

[3] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 263–272. ACM, Nov. 2005.

[4] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 135–145. ACM, Aug. 2000.

[5] R. Hodován and Á. Kiss. Practical improvements to the minimizing delta debugging algorithm. In *Proceedings of the 11th International Conference on Software Engineering and Applications (ICSOFT-EA 2016)*, page (to appear). SciTePress, July 2016.

[6] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[7] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 83–93. ACM, Nov. 2008.

[8] Microsoft Corporation. Security development lifecycle (verification phase). https://www.microsoft.com/en-us/sdl/default.aspx.

[9] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 142–151. ACM, May 2006.

[10] G. S. Misherghi. Hierarchical delta debugging. Master's thesis, University of California, Davis, June 2007.

[11] T. Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2013.

[12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 335–346. ACM, June 2012.

[13] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 30–39. IEEE, Oct. 2003.

[14] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[15] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*, pages 439–449. IEEE Press, Mar. 1981.

[16] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer-Verlag, Sept. 1999.

[17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.

[18] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 81–91. ACM, Nov. 2006.