Abstracting Failure-Inducing Inputs

Rahul Gopinath
rahul.gopinath@cispa.saarland
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Alexander Kampmann
alexander.kampmann@cispa.saarland
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Nikolas Havrikov
nikolas.havrikov@cispa.saarland
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Ezekiel O. Soremekun
ezekiel.soremekun@cispa.saarland
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Andreas Zeller
zeller@cispa.saarland
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

ABSTRACT

A program fails. Under which circumstances does the failure occur? Starting with a single failure-inducing input ("The input `((4))` fails") and an input grammar, the DDSET algorithm uses systematic tests to automatically generalize the input to *an abstract failure-inducing input* that contains both (concrete) terminal symbols and (abstract) nonterminal symbols from the grammar—for instance, "`(((`⟨*expr*⟩`)))`", which represents any expression ⟨*expr*⟩ in double parentheses. Such an abstract failure-inducing input can be used (1) as a *debugging diagnostic*, characterizing the circumstances under which a failure occurs ("The error occurs whenever an expression is enclosed in double parentheses"); (2) as a *producer* of additional failure-inducing tests to help design and validate fixes and repair candidates ("The inputs `((1))`, `((3 * 4))`, and many more also fail"). In its evaluation on real-world bugs in JavaScript, Clojure, Lua, and UNIX command line utilities, DDSET's abstract failure-inducing inputs provided to-the-point diagnostics, and precise producers for further failure inducing inputs.

CCS CONCEPTS

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

KEYWORDS

debugging, failure-inducing inputs, error diagnosis, grammars

ACM Reference Format:
Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Abstracting Failure-Inducing Inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397349

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07.
https://doi.org/10.1145/3395363.3397349

1 INTRODUCTION

Having to deal with software failures is the daily bread of software developers—frequently during development and testing, (hopefully) less so during production. Since failures are caused by concrete inputs, but must be fixed in abstract code, developers must determine the *set of inputs that causes the failure*, such that the fix applies to precisely this set. This is important, as an incorrect characterization leads to incomplete fixes and unfixed bugs.

As an example, consider a calculator program that fails given the input in Fig. 1a. To identify the *set of failure-inducing inputs*, the developer must ask herself: Is the error related to parenthesized expressions? *Any* parenthesized expression? *Doubled* parentheses? Or just *nested* parentheses? Or is the error related to addition, multiplication, or the combination of both? Each of these conditions induces a different set of inputs, and to fix the bug, the developer has to identify the failure-inducing set as precisely as possible—typically following the scientific method through a series of experiments, refining and refuting hypotheses until the failure-inducing set is precisely defined.

```
1 + ((2 * 3 / 4))
```
(a) Failure-inducing input

(((⟨*expr*⟩)))
(b) Abstract failure-inducing input

Figure 1: Input for the calculator program.

⟨*start*⟩ ::= ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*int*⟩ | ⟨*var*⟩
| ⟨*prefix*⟩ ⟨*expr*⟩ | "`(`" ⟨*expr*⟩ "`)`" | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
⟨*op*⟩ ::= " `+` " | " `-` " | " `*` " | " `/` "
⟨*prefix*⟩ ::= "`+`" | "`-`"
⟨*int*⟩ ::= ⟨*digit*⟩⟨*int*⟩ | ⟨*digit*⟩
⟨*var*⟩ ::= ⟨*chars*⟩⟨*char*⟩ | ⟨*chars*⟩
⟨*digit*⟩ ::= "`0`" | "`1`" | "`2`" | "`3`" | "`4`" | "`5`" | "`6`" | "`7`" | "`8`" | "`9`"
⟨*char*⟩ ::= "`a`" | "`b`" | "`c`" | "`d`" | "`e`" | "`f`"

Figure 2: Simple expression grammar

In this paper, we present an approach that fully *automates* the process of characterizing failure-inducing inputs. The DDSET algorithm[1] starts with a single failure-inducing input (such as the one above), a precise characterization of the failure and an input grammar (in our case, representing arithmetic expressions, as in Fig. 2). Using these three, it runs a number of experiments (tests) to derive *an abstract failure-inducing input* that abstracts over the original input, replacing concrete terminal symbols (characters) by abstract *nonterminals* from the grammar. In our example, such an abstract failure-inducing input produced by DDSET would be the one in Fig. 1b, where ⟨*expr*⟩ is actually *any* expression as given by its grammar expansion rule. Formally, an abstract failure-inducing input represents the set of all inputs obtained by expanding the nonterminals it contains—for Fig. 1b, *any* expression contained in double parentheses.

Can we determine that *all* expansions are failure-inducing? This would require a fair amount of symbolic analysis, and be undecidable in general. Instead, DDSET exploits the fact that the abstract failure-inducing input can be used as a *producer* of inputs. DDSET thus instantiates it to numerous concrete test inputs; the abstract failure-inducing input is deemed a valid abstraction only if *all* its instantiations reproduce the original failure.

The concept of an abstract failure-inducing input and how to determine it are the original contributions of this paper. Such an abstract failure-inducing input can be used for:

Crisp failure diagnostics. Abstract failure-inducing inputs such as `((`⟨*expr*⟩`))` precisely characterize the condition under which the failure occurs: "The failure occurs whenever an expression is surrounded by double parentheses". Their simplicity is by construction, as they are at most as long (in the number of tokens) as the shortest *possible input that reproduces the failure.* Should a developer prefer a number of examples rather than the abstraction, an abstract failure-inducing input can be used to produce *minimal inputs* similar to delta debugging [19]; in our case, this would be inputs such as `((0))`, `((1))`, etc.

Producing additional failure-inducing inputs. In manual debugging as well as automated repair, a common challenge is to *validate* a fix: How do we know we fixed the cause and not the symptom? An abstract failure-inducing input as produced by DDSET can serve as *producer* to generate several inputs that all trigger the failure. In our case, these would include inputs such as `((1))`, `((-3))`, `((5 + 6))`, `((8 * (3 / 4)))`, `(((9) * (10)))`, or `((11 / 12 + -13))`. All these inputs trigger the same original failure, and any fix should address them all. A badly designed fix that would cover only a subset (say, some single symptom of the original input such as "no digits with two levels of parenthesis") would be invalidated in an instant.

How does DDSET produce an abstract failure-inducing input? DDSET uses the given input grammar (e.g. Fig. 2) to *parse* the input into a *derivation tree*, representing the input structure by means of syntactic categories; Fig. 3 shows the derivation tree for our example. DDSET then applies the following steps:

[1] DDSET = Delta debugging for input sets

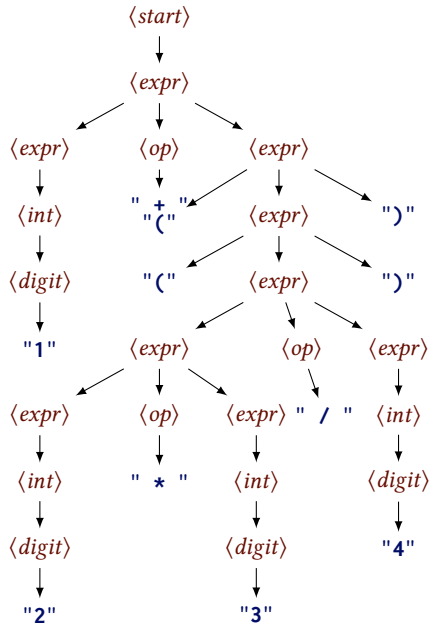

Figure 3: A derivation tree for `1 + ((2 * 3 / 4))`

(1) **Reduction.** DDSET first *reduces* the input to a minimal failure-inducing input. As the input grammar is available, we can make use of efficient syntax-based reduction [18] rather than lexical delta debugging [19]. For effective reduction, these algorithms require a precise test condition that either
   (a) identifies that the input produced was semantically invalid
   (b) identifies that the input while semantically valid, but, failed to reproduce the error, or
   (c) identifies that the input succeeded in reproducing the failure.

   In our case, the input `((3))` is the result of reducing `1 + ((2 * 3 / 4))`.

(2) **Abstraction.** Even given a simplified input such as `((4))`, we do not know which elements cause the failure—is it the first parenthesis, both of them, or the number 4? To determine this, DDSET makes use of the same test procedure used by the *reduction* step, DDSET uses this test procedure to determine which concrete symbols can be replaced by other input fragments while still producing failures. In our case, it turns out that in the derivation tree of `((4))` (Fig. 5), the symbol `4` can actually be replaced by any ⟨*digit*⟩, ⟨*int*⟩, and even ⟨*expr*⟩; any expansion still produces failures. As a result, we obtain the derivation tree in Fig. 6, which expands into the *concrete failure-inducing input.* Hence, we characterize the abstract failure-inducing input as `((<expr>))`.

(3) **Isolating Independent Causes.** Say the input was `((1)) + ((2 * 3)) - ((5))` and the failure cause was at least a pair of doubled parenthesis. Then, a naive abstraction will find the following fragments can be replaced by arbitrary expressions ⟨*expr*⟩ `+ ((2 * 3)) - ((5))`, `((1)) +` ⟨*expr*⟩ `- ((5))`, and `((1)) + ((2 * 3)) -` ⟨*expr*⟩ while still

inducing failures. This will lead to the erroneous conclusion that the final abstraction is ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩. Our *isolation* step will correctly extract ((⟨*expr*⟩)) ⟨*op*⟩ ((⟨*expr*⟩)) ⟨*op*⟩ ((⟨*expr*⟩)). Steps 2 and 3 work recursively until all independent causes are identified.

(4) **Sharing Abstraction.** Given an input as `a + a`, with the failure condition being the repetition of a variable, the previous *abstraction* and *isolation* steps will find that the first variable `a` cannot be replaced with another arbitrary variable. The *sharing abstraction* step identifies such parts and verifies that all such parts can together be replaced by a shared abstraction. The resulting abstract failure-inducing input ⟨*$var1*⟩ ⟨*op*⟩ ⟨*$var1*⟩ precisely identifies ⟨*$var1*⟩ as a string that is shared.

(5) **Handling Lexical Tokens.** A number of programming languages use *lexers* that skip over whitespace and comments. Further, program syntax may include optional elements such as annotations that are present in the derivation tree[2] and hence, the abstract pattern derived resolves to an empty string in the minimized input string. Finally, due to the way parsers and lexers work, tokens such as `begin` are represented by a nonterminal element such as ⟨*BEGIN*⟩, which can have only a single possible value. As these do not help developers, we identify these elements and replace them with their value in the compact representation.

The result of these steps is a **derivation tree** where *abstract*, *shared* and *invisible* nodes are marked, and a **compact representation** of this derivation tree as an abstract failure-inducing input. The compact representation is useful for developers while the derivation tree can be used as a producer for inputs.

In all that, the abstract failure-inducing inputs produced by DDSET capture dependencies even for complex input languages. Fig. 4a shows a failure-inducing input for the *Rhino* JavaScript interpreter. Fig. 4b shows its abstract representation as produced by DDSET. We see that instead of `baz`, we can have *any* identifier (as long as it is shared) and *any* variable declaration. The instantiations of this abstraction can produce numerous test cases that all help ensuring a proper fix.

In the remainder of this paper, we follow the steps of our approach, detailing them with the calculator example. After introducing central definitions (Section 2), we detail the steps of DDSET, namely reduction (Section 3), abstraction (Section 4), isolation (Section 5) identifying shared parts (Section 6), and identifying invisible elements (Section 7). We discuss properties and limitations of DDSET (Section 9). In our evaluation (Section 10), we apply DDSET on a range of bugs and subjects, assessing its effectiveness. After discussing threats to validity (Section 11) and related work (Section 12), Section 13 closes with conclusion and future work.

2 DEFINITIONS

In this paper, we use the following terms:

Input. A contiguous sequence of symbols fed to a given program. That is for our example, `1 + ((2 * 3 / 4))`, the symbols `"1" " + " "(" "(" "2" " * " "3" " / " "4" ")" ")"` form our input.

[2]These are marked as *SKIP* in ANTLR grammars.

Alphabet. The *alphabet* of the input accepted by a given program is the set of all non divisible symbols that the program can accept. In our example, the digits (`"0" to "9"`), operators (`" + " " - " " * " " / "`), prefixes (`"+" "-"`) and parenthesis (`"(" ")"`) forms the alphabet.

Terminal. An input symbol from the alphabet. These form the leaves of the derivation tree. For example, `"2"` is a terminal and so is `" + "`.

Nonterminal. A symbol outside the alphabet that has a grammar definition. These form the internal nodes of the derivation tree. From our example, ⟨*expr*⟩ is one of the nonterminals.

Context-Free Grammar. A set of recursive rules that is used to describe the structure of input. The context-free grammar is composed of a set of nonterminals and corresponding definitions that define the structure of the nonterminal. Each definition consists of multiple rules that describe alternative ways of defining the nonterminal. Fig. 2 describes a context-free grammar for an expression language.
We assume that the context-free grammar is given, and that it can parse the input to a[3] *derivation tree*[4].

Rule. A finite sequence of terminals and nonterminals that describe an expansion of a given nonterminal. For our example, ⟨*expr*⟩ ::= ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ is one of the rules that defines the nonterminal ⟨*expr*⟩ in the grammar.

Derivation Tree. A *derivation tree* is an ordered tree that describes how an input string is parsed by the rules in the grammar. Fig. 3 shows a derivation tree built by parsing `1 + ((2 * 3 / 4))` using the context-free grammar in Fig. 2.

Predicate. A test *predicate* determines if the given input is able to reach and reproduce the failure condition. If the input was not legal, that is, the input is invalidated by checks on the input before the predicate is reached, then the result is semantically UNRESOLVED. If the input reaches the predicate, and the failure condition is reproduced, the result is FAIL. If not, the result is PASS. Note that at all times, inputs are syntactically valid—that is, each input conforms to the context-free grammar. These follow the original definitions [19].

Compatible Node. A node is *compatible* to another if both have the same nonterminal. E.g. the root node for the expression `2 * 3` is an ⟨*expr*⟩. Similarly, the root node for the expression `2 * 3 / 4` is also an ⟨*expr*⟩. Hence, these two nodes are compatible.

Compatible Tree. A tree is *compatible* to another if both have compatible root nodes.

Generated Compatible Tree. One may randomly *generate compatible trees* given a nonterminal by the following production process:

(1) Stochastically choose one of the rules from the definition corresponding to the nonterminal. The terminals and nonterminals in the rule form the immediate children of the root node of the generated tree.

[3]We assume that there is one canonical derivation tree. That is, no ambiguity, or in the case of ambiguity, and all derivation trees are semantically valid, we simply choose one tree. If not all derivation trees are semantically valid, we assume that there exists a procedure to identify the correct tree.

[4]If not, there are two choices: fix the input or the grammar. A technique like lexical *ddmax* would isolate the failure-inducing input in context and give sufficient hints to debug the input or the grammar such that the input can be parsed.

```
1 var {baz: baz => {}} = baz => {};
```

(a) Failure-inducing input

```
var {⟨$Id1⟩:⟨$Id1⟩ => {} } = ⟨variableDeclaration⟩;
```

(b) Abstract failure-inducing input for Fig. 4a

Figure 4: Issue 385 of the Rhino JavaScript interpreter.

(2) For each nonterminal in the rule, stochastically choose a rule from the corresponding definition in the context-free grammar.
(3) Continue the process until no nonterminals are left.

Generated String. Given a randomly generated compatible tree, the corresponding string representation is called the *generated string*. There can be an infinite number of generated strings corresponding to a nonterminal if the nonterminal is recursive (i.e the nonterminal is reachable from itself).

Reachable Nonterminal. A nonterminal a is *reachable* from another nonterminal b if a is reachable from any of the rules in the definition of b. A nonterminal is reachable from a rule if
(1) that nonterminal is present in the rule or
(2) that nonterminal is reachable from any of the nonterminals in the rule.

From our expression grammar, the nonterminal⟨*op*⟩ is reachable from the nonterminals ⟨*expr*⟩ and ⟨*start*⟩. The nonterminal ⟨*expr*⟩ is reachable from itself and ⟨*start*⟩.

Subtree. For any given node in a tree, a *subtree* refers to the tree rooted in any of the reachable nonterminals from that node.

String Representation. For any given subtree, the corresponding *string representation* is the string fragment from the original input that corresponds to the subtree.

Compact Representation. For any given tree, the corresponding *compact representation* is the abstract string representation where the string representation of nodes marked as abstract are the corresponding nonterminal, shared nodes are represented by a parametrized nonterminal, and invisible nodes are represented by their corresponding string.

1-minimal Input. An input string is *1-minimal* if the predicate indicates that the string causes the failure, and removing any one symbol from the input no longer causes the failure.

1-minimal-tree Input. An input has a *1-minimal-tree* as its derivation tree if there is no node in the derivation tree that can be further simplified by the given reducer. Note that the definition of *1-minimal-tree* is dependent on the reduction algorithm used.

3 REDUCTION

Let us now get into the details of DDSET. As discussed in Section 1, DDSET starts with *reducing* the given input to a minimal input. As a reminder, as input to DDSET, we have

- the *predicate*—in our case, a program that fails whenever there are doubled parentheses in the input. For the sake of simplicity, we assume that the "calculator" program simply matches the input against a regular expression

  `/.*[(][(].*[)][)].*/`

  —that is, it will fail on any input that contains a pair of opening and closing double parentheses, and pass otherwise.
- the *failure-inducing input*—in our case, the input

  `1 + ((2 * 3 / 4))`
- the *input grammar*, as shown in Fig. 2.

DDSET uses the *Perses* reducer from Sun et al. [18]. We provide a brief overview of the Perses reducer. We first parse the input using the grammar provided, which results in a derivation tree (Fig. 3). The reduction algorithm (Algorithm 1) accepts this derivation tree and minimizes it to a minimal tree (Fig. 5).

Algorithm 1 The Perses reduction algorithm

```
function REDUCTION(string, grammar, predicate)
    dtree ⇐ parse(string, grammar)
    p_q ⇐ priority_queue((dtree, ∅))
    while p_q ≠ ∅ do
        dtree, path ⇐ p_q.pop()
        snode ⇐ dtree.get(path)
        trees ⇐ ∅
        compatible_nodes ⇐ snode.get_all_nodes(snode.key)
        if ∅ ∈ grammar[snode.key] then
            compatible_nodes.append(∅)
        end if
        for node ∈ compatible_nodes do
            ctree ⇐ dtree.replace(path, node)
            if predicate(ctree.to_s) == FAIL then
                trees.append((ctree, path))
            end if
        end for
        if trees ≠ ∅ then
            tree ⇐ minimal(trees)
            p_q.insert(tree, path)
        else
            for child ∈ nonterminals(snode.children) do
                p_q.insert(dtree, child.path)
            end for
        end if
    end while
    return dtree
end function
```

We start by maintaining a priority queue of tuples. The first item in the tuple is a derivation tree and the second item in the tuple is a path to a particular node in that derivation tree. We next add the full derivation tree and a path to the root node (t_0, p_0) as the first item in the priority queue. The tree priority is determined by the number of terminal symbols (leaf nodes) on the derivation tree, followed by number of terminal symbols on the subtree that is indicated by the path. That is the shortest token sequence is at the top of the priority queue.

Next, DDSET performs the following steps in a loop.

(1) Extract the top tuple. It contains a complete derivation tree and the path to a subtree from that tree.

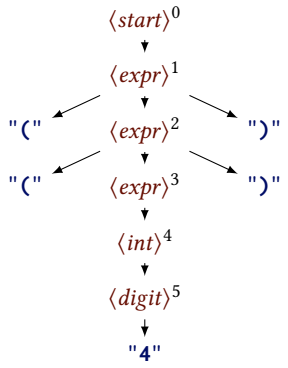

Figure 5: The derivation tree for `((4))`. Nodes are annotated with numbers for easier reference.

(2) Given the subtree, identify the *compatible reachable nodes* from the root node of the subtree, ordered by their depth (shallowest first). These are *alternative* trees that we can replace the current subtree with a high chance of reproducing the failure. If the grammar allows the current node to be empty, then an empty node is added to the compatible nodes with a depth 0.
(3) For each tree in the alternative, replace the current tree with the tree in the alternative producing a new derivation tree. Collapse the new derivation tree to its corresponding string representation, and check if the predicate confirms reproducing the failure. Collect every such alternative tree, and identify the tree that produces the smallest input (`minimal()`). Generate tuples for this tree — the first element is the derivation tree corresponding to the new input string, and the path is the same as the current path. Add this tuple to the priority queue. If we could find a smaller input in this step, go back to the first step.
(4) If no smaller inputs could be found, generate new tuples by using the same derivation tree, but with paths that correspond to the children of the current node. Add the new tuples to the priority queue. Then go back to the first step.
(5) The loop ends when the priority queue is empty.

At this point, we will have a minimal input where the predicate reproduces the failure.

4 ABSTRACTION

Abstraction is the process of identifying the causative parts that contributed directly to the failure observed, and identifying and abstracting the non-causative parts. For abstraction, the idea is to identify which parts of the given derivation tree are required to produce a failure, and which parts can be replaced by a random generated string.

The algorithm is as follows. We start with the derivation tree that corresponds to the input string.

(1) Identify the nonterminal of the top node of the tree.
(2) Generate N random generated trees that correspond to the nonterminal where N is user configurable, and determined by the accuracy desired.

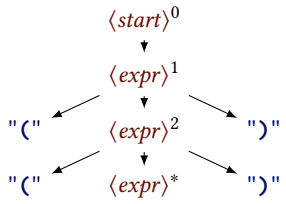

Figure 6: The abstract derivation tree for `((4))`. The abstract nodes are marked with '*'.

(3) For each generated tree, obtain the full input string from the full derivation tree where the current tree is replaced by the generated tree.
(4) For each such string, check if predicate responds with FAIL. If for any generated tree, the predicate responds with PASS, then the tree cannot be abstracted. If *all* checked alternatives result in FAIL, we assume the tree can be abstracted. If predicate responds with UNRESOLVED, then the input failed to validate. We can only make the distinction between concrete and abstract using semantically valid inputs. Hence, we ignore the input—that is, we do not consider this input among the `N` random inputs, and generate a new input to try again.
(5) If the tree can be abstracted, add the path to this node to *abstract nodes*.
(6) If the tree cannot be abstracted, then continue the same procedure with its nonterminal children.

Using our example derivation tree at Fig. 5, we first consider abstracting $\langle start \rangle^0$. The generated strings for $\langle start \rangle^0$ contain strings such as `100`, `2 + 3` etc. These result in PASS from the predicate. That is, $\langle start \rangle^0$ cannot be abstracted, and hence, marked as *concrete*. Similarly, $\langle expr \rangle^1$, also cannot abstracted, and marked as concrete. Considering $\langle expr \rangle^2$, all generated strings produced are parenthesized. However, only a few are double parenthesized. Hence, $\langle expr \rangle^2$ is also marked as concrete. Considering $\langle expr \rangle^3$, all generated strings produced are of the form `/((.*))/` which successfully triggers the predicate. Hence, the path to $\langle expr \rangle^3$ is added to *abstract nodes*. This results in the abstract derivation tree in Fig. 6.

This sequence of steps produces an annotated derivation tree where each node is marked as either abstract or concrete. This abstract derivation tree encodes what exactly caused the failure. In our case, the abstract derivation tree generates the string $((\langle expr \rangle))$, which clearly suggests the reason for failure—doubled parentheses.

5 ISOLATING INDEPENDENT CAUSES

Our abstraction algorithm introduced so far can fail when the failure is caused by interaction of multiple syntactical elements. For example, here is a minimized input string [7] for the *closure* JavaScript compiler: `{ while ((l_0)) { if ((l_0)) { break;;var l_0;continue }0 } }` . The problem here is that one requires exactly two instances of `l_0` to reproduce the bug with the remaining element allowed to be any syntactically valid identifier. So our first algorithm will try replacing the first `l_0`, and succeed because the other two instances are present in the string. Similarly, the second and third `l_0` will also be identified as abstractable because

the first and second respectively are present in the strings generated. However, the abstraction generated `{ while ((`⟨*Id*⟩`)){ if ((`⟨*Id*⟩`)) { break;;var `⟨*Id*⟩`;continue }0 } }` is incorrect — the failure would not be reproduced if all three are replaced by separate identifiers.

To address this problem, we verify that each of the nodes identified as abstract on its own, continued to do so when other abstract nodes are also replaced by randomly generated values. If the current node is no longer abstract when other abstract elements are replaced by randomly generated values, we unmark the current node as abstract, and run the abstraction algorithm on the child nodes, but with other abstract nodes replaced with random values.

The modified algorithm is given in Algorithm 2. The `generate()` function when given a derivation tree, and a set of paths, generates a new tree with all nodes pointed to by the paths replaced by a random compatible node.

We note that isolating independent causes is a well known problem for variants of delta debugging [15, 16], and our algorithm provides a solution if applied directly to non-reduced input (under the constraint that the faults can be isolated to separate parts of the input). That is, if the input to the algorithm contains two separate faults, both faults will be concretized and retained in the output.

6 IDENTIFYING SHARING ABSTRACTION

The abstraction algorithm detailed before works well when the examined parts are independent. However, complex languages often have elements such as variable definition and references that should be changed together.

As an example, Fig. 4a shows a failure-inducing input for the Rhino JavaScript interpreter [13]. There are a few questions that the developer may wish to answer here based on this fragment:

(1) We see two empty `{}`. Do these need to be empty?
(2) What exactly about the empty pattern is important here?
(3) Can they be other data structures?
(4) Is it required that the patterns are exactly the same?
(5) Can one reproduce the failure with a variable other than `baz`?
(6) Do the three `baz` need to the exact same variable?

To further abstract such an input, we need to abstract over variable names—but in a *synchronized* fashion. To identify such patterns, we start with the abstract derivation tree, and first identify all nonterminals that are present in the derivation tree. Next, for each nonterminal, we identify its nodes in the derivation tree, and group the nodes based on the corresponding string representation. For any such grouping which contains more than one node, we generate compatible trees for that nonterminal and replace all the nodes in that group with that tree. We then obtain the string representation of the complete derivation tree and verify if the new string reproduces failure when the predicate is applied. If all such generated strings can reproduce the failure, the particular nodes under this grouping are parametrized by prefixing their nonterminal with a "`$`" and suffixing it with a unique id for this group.

For example, say we have the following input `int v=0; v=1/v` which induces an error (Fig. 7). The *abstraction* step will be unable to abstract `v`, and nodes annotated *e*, *k* and *q* and their parent nodes will be left concrete. Next, we look at the string representations of

Algorithm 2 The abstraction and isolation algorithms

```
function CAN_GENERALIZE(tval, dtree, grammar, predicate, rnodes)
    checks ⇐ 0
    while checks < MAX_CHECKS do
        newtree ⇐ generate(dtree, grammar, [tval] + rnodes)
        pres ⇐ predicate(newtree.to_s)
        if pres = PASS then
            return false
        else
            if pres = FAIL then
                checks = checks + 1
            end if
        end if
    end while
    return true
end function
function ABSTRACTION(tval, dtree, grammar, predicate, rnodes)
    path, status ⇐ tval
    if dtree.get(path).is_terminal then
        return []
    end if
    abstract ⇐ can_generalize(tval, dtree, grammar, rnodes)
    if abstract then
        if status = UNCHECKED then
            return [(path, UNVERIFIED)]
        else
            return [(path, VERIFIED)]
        end if
    else
        paths ⇐ ∅
        for child ∈ nonterminals(snode.children) do
            tval ⇐ (child.path, UNCHECKED)
            paths.extend(abstraction(tval, dtree, predicate, rnodes))
        end for
        return paths
    end if
end function
function ISOLATION(tree, grammar, predicate)
    unv ⇐ [(∅, UNCHECKED)]
    verified ⇐ ∅
    original ⇐ ∅
    while unv ≠ ∅ do
        v ⇐ unv.shift()
        notpv ⇐ filter(original, λo :!o.isparent(v))
        topnodes ⇐ filter(notpv, λo :!ischildofany(o, notpv))
        runverified ⇐ filter(unv, λo :!ischildofany(o, topnodes))
        rnodes ⇐ runverified + topnodes
        newpaths ⇐ abstraction(v, tree, grammar, predicate, rnodes)
        for p ∈ newpaths do
            if p[1] == VERIFIED then
                verified.append(p)
            else
                unv.append(p)
            end if
        end for
        original.append(v)
    end while
    return mark_verified(tree, verified)
end function
```

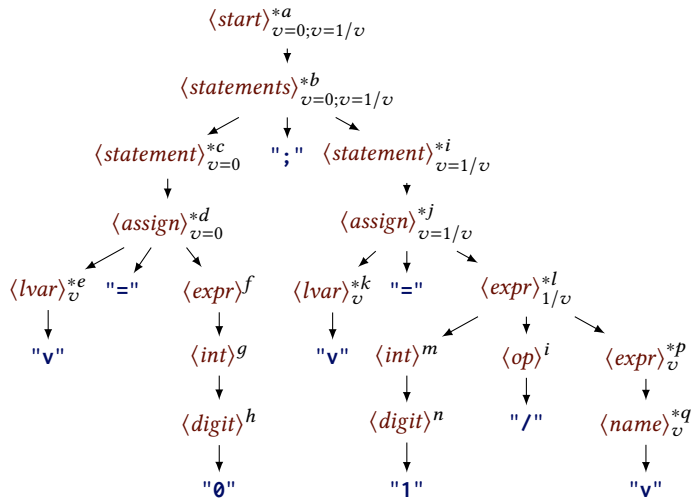

Figure 7: A derivation tree for `int v=0; v=1/v`. The concrete nodes are annotated with * and the string representation is provided next to these nodes.

each nonterminal. For example, the node d has a string representation `v=0` while k, e, p, and q has the same string representation `v`. We only collect string representations of concrete nodes (annotated by * in Fig. 7). From the figure, we can see that the nodes a and b have similar string representation — `v=0;v=1/v` similarly, c and d have `v=0`, and e, k, p, and q have `v` and so on. Out of these, we eliminate nodes that are a child of any of the other nodes in the grouping. This eliminates b since it is a child of a, which also eliminates the grouping since there is only one member left in the group.

This leaves us with one group, with members e, k, and p corresponding to `v`. Next, we pick a randomly generated value for one of the nodes; Say `x`, and use this value in place of each nodes, generating a new input string `x=0;x=1/x`. This string is now checked to see if it induces the same failure. (Only values that are legal to be replaced in each node is used, and values that result in UNRESOLVED are discarded). If the randomly generated value successfully reproduces the failure (as would happen in this case), we repeat the procedure for a fixed number of times for statistical confidence. If every such legal string induces the given failure, we mark the set of nodes as shared.

What if we have a set of nodes with the same string representation, but fails to vary together? E.g say we have an input `myval="myval"; check(myval)`. The input produces a failure if a variable with value `"myval"` is passed to the function `check()`. Here, `myval` can be replaced by any legal variable name so long it is correctly passed to `check()`. However, the string value has to remain `"myval"`. To handle such cases, we produce combinations of nodes which are sorted by the number of nodes. That is, all nodes are checked for sharing first. Then, combinations with one node excluded are tried next, and then combinations with two nodes excluded and so on (denoted by `len_sorted_combinations()`). The first set of nodes that can vary together is chosen for a shared name. If no such set is found, the entire grouping is discarded. The complete algorithm is formalized in Algorithm 3.

Algorithm 3 Identify shared nodes

```
function IDENTIFY_SHARED _NODES(tree, grammar, predicate)
    cpaths ⇐ concrete_paths(tree)
    snodes ⇐ find_similar(tree, cpaths)
    mtree ⇐ tree
    for key ∈ snodes do
        plst ⇐ snodes[key]
        paths ⇐ find_shared(plst, grammar, mtree, predicate)
        if paths then
            mtree = mark_context_sensitive(paths, mtree)
        end if
    end for
    return mtree
end function
function CONCRETE_PATHS(node)
    if node.abstract then
        return []
    end if
    my_paths ⇐ [node.path]
    for cnode ∈ node.children do
        my_paths = my_paths + concrete_paths(cnode)
    end for
    return my_paths
end function
function FIND_SIMILAR(tree, paths)
    strings ⇐ ∅
    for path ∈ paths do
        node ⇐ tree.get(path)
        s ⇐ node.to_s
        if s.len = 0 then
            continue
        end if
        strings[(node.key, s)].add(path)
    end for
    res ⇐ filter(strings, λs : strings[s].len() > 1)
    return res
end function
function FIND_SHARED(plst, grammar, tree, predicate)
    for paths ∈ len_sorted_combinations(plst) do
        checked = 0
        while checked < MAX_CHECKS do
            val ⇐ generate(tree, paths, same = true)
            res ⇐ predicate(val)
            if res = PASS then
                break
            else
                if res = FAIL then
                    checks = checks + 1
                end if
            end if
        end while
        if checked == MAX_CHECKS then
            return paths
        end if
    end for
    return ∅
end function
```

The result of this step is the abstract failure-inducing input shown in Fig. 4b. With this, the questions that we asked above are immediately answered. First, we avoid ambiguity. The abstract failure-inducing input clearly indicates that all that is required is for the first two ⟨*Id*⟩ to be the same identifier (baz in the minimized string) and the first `{}` to be present.

Second, the abstract failure-inducing input suggests that the token ⟨*variableDeclaration*⟩ is effectively a space filler, as any instantiation of ⟨*variableDeclaration*⟩ would do. It further indicates precisely what portion of the processing program contributed to the failure—something related to the processing of the syntactical elements ⟨*Id*⟩ and ⟨*variableDeclaration*⟩. This is an important information which was not visible in the minimized input fragment. In fact, this addresses a major drawback of current HDD variants as pointed out by Regehr et al. [16] (*generalized transformations*) and Pike [15] (*sharing*).

7 HANDLING LEXICAL TOKENS

In our last step, we tackle the distinction between *lexical* and *syntactical* features. Input strings often contain lexical parts such as whitespace and comments. These are often skipped over before parsing. These elements hence do not contribute to the program semantics. Showing that there is an abstract whitespace in between any two elements hence is of little help to the developer who interprets the abstract failure-inducing input.

Further, the abstract failure-inducing input may contain nonterminal symbols that represent optional parts of the grammar. These can be parts such as type hints in Python, documentation annotations, optional arguments to command lines that can be entirely skipped etc. While these are important sources of variability when the abstract failure-inducing input is used as a producer, a developer may not care about the existence of these nonterminals. Hence, we categorize them as *invisible* elements, and remove them from the compact representation of the abstract failure-inducing input.

Finally, tokens such as `begin` and `if` may be represented by nonterminal symbols such as ⟨*BEGIN*⟩ and ⟨*IF*⟩. Since marking them as abstract does not provide any value-add for the developers, we identify such lexical tokens and replace them with their values in the compact representation.

8 THE COMPLETE DDSET ALGORITHM

Algorithm 4 shows how all the components fit together. The initial input is first parsed using a grammar, which results in a derivation tree. This derivation tree is passed to the `reduction` function, which minimizes the derivation tree to a *1-minimal-tree* input. The *1-minimal-tree* is then passed to the `isolation` function. The `isolation` function in turn, uses `abstraction` to identify nodes that are independently abstract. Finally, the function `identify_shared_nodes` identifies and marks parts that are shared. The tree thus produced is converted to a string representation and returned.

9 PROPERTIES AND LIMITATIONS

Our DDSET approach has a number of interesting properties and limitations, which we list here.

Algorithm 4 Top level

```
function GET_ABSTRACTION(grammar, myinput, predicate)
    mtree ⇐ parse(myinput, grammar)
    rtree ⇐ reduction(mtree, grammar, predicate)
    itree ⇐ isolation(rtree, grammar, predicate)
    ctree ⇐ identify_shared_nodes(itree, grammar, predicate)
    return compact_rep(ctree)
end function
```

Approximation. In general, an abstract failure-inducing input will be an approximation. The reason is that a context-free grammar cannot fully characterize a universal grammar (that is, a Turing machine). There are specific causes for approximation:

(1) During abstraction, if we are unable to generate any semantically valid inputs, we give up, and mark the node as concrete. This does not mean that the node cannot be generalized. Further, there may be other syntactical patterns that produce the same bug, which is not captured in the particular abstract failure-inducing input we derive from the minimized input string. Both of these are causes for *underapproximation.*

(2) We have a fundamental limitation in that we rely on randomness to generate possible substitutes for particular nodes, and it is possible that the random strategy was unable to provide a counterexample in the time budget allotted. In such cases, we may erroneously mark nodes as abstract when they are not and *overapproximate.* This risk can be limited by running more tests.

Limitations of Delta Debugging. Our approach inherits limitations from delta debugging, such as reliance on a precise test case [16] that may require knowledge about internal program structure (such as a crash location). We also require that the test case clearly distinguishes between FAIL and UNRESOLVED, especially as several of the generated test inputs may be valid syntactically, but invalid semantically. Finally, we require that the test case be deterministic—that is, the program behavior fully depends on the given input.

Grammar Quality. Like Perses and other HDD variants, DDSET relies on a grammar to reduce and abstract inputs. Since it uses the grammar both for decomposing as well as for producing inputs, it is important to have a high-quality grammar. If the grammar is too lax, it will parse inputs well; many ANTLR parsers err on this side. However, many of the *produced* inputs will be invalid, requiring the test to strictly separate between FAIL and UNRESOLVED. If the grammar is too tight, opportunities for generalization will be missed.

Performance. We note that DDSET is computationally expensive. In particular, in the worst case, each node in the derivation tree contributes to N executions of the test case where N is dependent on the desired accuracy. The number of nodes in a derivation tree depends on the branching factor, and can be approximated to $O(nlog(n))$ where n is the number of tokens, and the base of the logarithm is the branching factor. Furthermore, DDSET may need to generate a number of syntactically inputs for generating a single semantically

valid input. If it required K syntactically valid inputs for one semantically valid input, the worst case runtime complexity of DDSET would be $O(K \times N \times n \times log(n))$ where n is the number of tokens in the input. The number of tests, however, is in line with Delta Debugging on complex inputs; as DDSET would typically be started automatically after an automated test has failed, there is no human cost involved.

At least as good as Delta Debugging. Despite the above limitations, let us point out that DDSET is at least as good as the state of the art. Notably, its *abstract failure-inducing inputs are never longer than the reduced failure-inducing input.* We always start with the minimized input, and abstraction only substitutes character sequences with single tokens. Hence, the length of the result (counting each token as a single element) will never be more than the length of the original minimized input. If there exists a better variant for HDD than Perses, we can simply use that variant instead.

10 EVALUATION

To evaluate DDSET, we pose the following research questions.

(1) **RQ1** How effective is the DDSET algorithm in *generating abstract failure-inducing inputs*? That is, we want to know if DDSET algorithm can accurately identify abstractable patterns in the given input, and whether identifying these patterns can lead to an overall reduction in the complexity of the input.
(2) **RQ2** How *accurate* are the patterns generated by the DDSET algorithm? That is, did the algorithm *correctly* identify parts that can be abstracted? Or were some parts mislabelled as abstract?

10.1 Evaluation Setup

For our evaluation, we used subjects for four input languages, from programming languages to command lines:

Javascript is a large language with numerous parser rules, keywords, and other special context rules. We used the Javascript grammar definition from the ANTLR project [2], which corresponds to the ECMAScript 6 standard. The ANTLR Javascript grammar contains both lexical (lexer) and syntactical (parser) specification. We used the following Javascript interpreters in the evaluation:
- Closure interpreter v20151216, v20200101, and v20171203. The bugs were obtained from the Closure issues page [8].
- Rhino interpreter version 1.7.7.2. The bugs for this project were obtained from the Github issues page [14].

Clojure[5] is a Lisp-like language with a limited set of parse rules that describe the main language. We used the Clojure grammar definition published by the ANTLR project [1] in 2014, which still parses later versions of Clojure. The following version was used for evaluation:
- Clojure interpreter version: 1.10.1. The bugs for this project were obtained from Clojure JIRA [6].

Lua[6] is a smaller language with a limited set parsing rules. We used the Lua grammar definition from the ANTLR project [3]. The following version was used for evaluation:
- Lua interpreter 5.3.5. The bugs for this project were obtained from the project page [11].

UNIX command line utilities. For the UNIX command line utilities, we chose the two commands *find* and *grep* which were published in the *DBGBench* [4] benchmark. The grammars for *find* and *grep* command line options were extracted from the manual pages. These grammars list which parts of the command line are optional, and which parts accept arguments. The syntax for arguments is also represented and includes regular expressions and file names. The particular coreutil bugs are identified by their hashes in DBGBench.

We converted each grammar from the ANTLR format to a pure context free form by extracting *optional* and *star* patterns to separate grammar rules. For each bug, we read the bug report, and identified the *smallest* input that was provided in the bug report by the reporter or later commenter. Using this input, we translated the bug behavior to a test case with the following properties:

- Successfully identify when the fault is triggered (FAIL).
- For complex languages (those except coreutils) the test case should identify whether the semantic rules of the language were fulfilled (PASS).
- Similarly, for complex languages, the test case should be able to accurately identify when the semantic rules are violated, leading to a rejection of the input (UNRESOLVED). This is because, for languages such as Clojure, Javascript and Lua, there is often a second stage after parsing where the program is statically analyzed to identify errors. Hence, while according to the grammar of Clojure, any parenthesized expression can be placed anywhere, a developer normally expects a specific kind of expression under, say, a parameter definition. For coreutils, the UNRESOLVED indication was not used.
- Triggering the timeout (one minute) was counted as PASS.

10.2 RQ1: Effectiveness of Abstraction

The effectiveness of abstraction aims to capture how effective DDSET is in identifying non-essential filler parts. To measure the effectiveness of abstraction, we assess the number of nonterminal symbols that could be used in a given minimal string. This indicates the amount of abstraction possible in that a nonterminal can be replaced by any of its semantics conforming expansion. The remaining characters in the string indicates what could not be abstracted in this way, and indicates the limit of DDSET.

Our results for abstraction are shown in Table 1:

- The **Bug** column contains the bug identifier in the particular bug tracking system the language uses. "rhino 385", for instance, denotes the bug in Fig. 4a.
- The **# Chars in Min String** column reports the length of the input string after it was minimized through the *perses* reduction algorithm. This is our starting point.
- The **# Visible Nonterminals** column reports the number of distinct nonterminals found by the abstraction algorithm that are visible to the user in the abstract failure-inducing input— the more nonterminal symbols found, the better the abstraction was, and the lower the cognitive load for the developer. In the pattern for "rhino 385", shown in Fig. 4b, we have three visible nonterminals.

Note that the Nonterminal count does not include number of nonterminal symbols for space, comments, and other lexically skipped parts of the program that are not part of the semantics. Any space left after minimization is counted as part of remaining characters. Secondly, any abstractable element that resulted in an empty string is skipped.

- The # **Invisible Nonterminals** column denotes the number of nonterminal symbols that are invisible (because they represent empty string or skipped space and comments). We see that the abstract failure-inducing input in Fig. 4b has 17 invisible nonterminals, denoting the spaces between elements.
- Within the nonterminals, the number of shared nonterminals is provided in the column # **Shared**. An item such as 2 + 3 indicates that two distinct shared nonterminals were found, of which the first had a two references, while the second had three references—the more such shared symbols found, the better (even better than nonterminals found) as it indicates areas of semantic importance, and hence, possible places that the developer should focus on. For Fig. 4b, this item is 2, which represents the single shared nonterminal (⟨*$Id1*⟩) which appears twice. For *lua-5.3.5.4*, this is 3 + 2 which indicates one shared nonterminal that repeats thrice, and another that repeats two times.
- The # **Remaining Chars** column reports the number of characters left after removing the tokens—for Fig. 4b, these are 7.
- Finally, the # **Executions** column reports the number of executions required for the complete abstraction (including minimization). It takes 14,015 executions of Rhino to obtain the abstract failure-inducing input in Fig. 4b. We note that the large number of executions is due to the high accuracy desired, and the accuracy desired can be controlled by the user.

In total, Table 1 shows that all 22 buts could be abstracted to varying degrees. In particular, nine bugs had a number of shared elements which should get extra attention from the developer. Any of the abstract failure-inducing inputs stands for an infinite set of inputs, which can all be used to generate more tests. Finally, the table shows that DDSET never does worse than the *minimized input* it started with.

DDSET could provide abstract failure-inducing inputs for all 22 bugs studied.

While the number of executions may seem high, it is in line with expectations (See "Performance" in Section 9) and the state of the art. It is also high because we used up to 100 test runs to validate each abstraction step. Reducing this number to, say, 10, would much increase performance, but also increase the risk of inaccuracy.

Users can choose between high accuracy and a lower number of executions.

Table 1: The effectiveness of abstraction on reported bugs

Bug	# Chars in Min String	# Visible Nonterminals	# Invisible Nonterminals	# Shared	# Remaining Chars	# Executions
lua-5.3.5 4	83	5	54	3+2=5	28	19265
clj-2092	55	1	24	=0	11	505
clj-2345	34	1	12	=0	4	911
clj-2450	185	5	60	2=2	27	1954
clj-2473	29	2	18	=0	10	5118
clj-2518	30	0	20	=0	8	741
clj-2521	135	2	23	=0	10	864
closure 1978	84	10	37	3=3	11	17174
closure 2808	14	3	7	2=2	1	167
closure 2842	60	6	41	3+3=6	14	551
closure 2937	35	2	21	=0	7	8203
closure 3178	42	7	26	2=2	8	14853
closure 3379	16	0	11	=0	4	935
rhino 385	33	3	17	2=2	7	14015
rhino 386	16	2	13	2=2	6	557
grep 3c3bdace	37	1	2	=0	31	236
grep 54d55bba	64	1	1	=0	61	239
grep 9c45c193	21	2	1	2=2	19	117
find 07b941b1	16	1	4	=0	15	280
find 93623752	11	1	3	=0	10	192
find c8491c11	15	1	3	=0	14	200
find dbcb10e9	15	2	3	=0	13	381

10.3 RQ2: Accuracy of Abstraction

The abstractions provided by DDSET are only useful if developers can be confident that they are *accurate*—that is, they are correct abstractions for a multitude of possible instantiations. Hence, we also judge the accuracy of our abstractions by generating random inputs from the produced abstract failure-inducing input, and checking how effective it was in generating failure-triggering inputs. Given that we are evaluating the effectiveness of the abstract failure-inducing input as a producer, we do not have to worry about the cognitive load of the developer. Hence, *invisible* nonterminals (i.e optional elements) are allowed during production.

For generating strings, we simply chose a random expansion for each of the abstract patterns. The complete string thus produced is evaluated using the test case.

The invalid values are produced because while generalizing, we filtered out produced invalid values from a given nonterminal. This means any nonterminals we use can produce invalid values. For example, consider the Closure bug 2937. The minimized string is `var A = class extends (class {}){}` and the abstraction produced is `var` ⟨*assignable*⟩ `= class extends (class`⟨*classTail*⟩`){}`. According to the Javascript grammar, ⟨*assignable*⟩ can be any ⟨*objectLiteral*⟩,

Table 2: Test generation from abstracted patterns

Bug	VALID %	FAIL %	Bug	VALID %	FAIL %
clj-2092	100	100	closure 3379	76	100
clj-2345	100	100	lua-5.3.5 4	100	100
clj-2450	62	100	rhino 385	49	100
clj-2473	40	100	rhino 386	100	100
clj-2518	100	100	grep 3c3bdace	100	100
clj-2521	100	100	grep 54d55bba	100	100
closure 1978	76	100	grep 9c45c193	100	100
closure 2808	100	100	find 07b941b1	100	100
closure 2842	100	99	find 93623752	100	100
closure 2937	36	100	find dbcb10e9	100	100
closure 3178	57	100	find c8491c11	100	100

which through a long chain of rules allows ⟨*singleExpression*⟩, which allows almost all other parts of Javascript to be present. However, only a limited number of items are allowed in the ⟨*assignable*⟩ position. This validation is handled external to the parser. Unfortunately, given that this validation is absent for the fuzzer, it produces inputs of the type: `{var { T[yield] (){return}} = class extends (class {}){}` where the fragment `{ T[yield] (){return}}` is allowed by the grammar, but invalid as an ⟨*assignable*⟩ according to the language semantics.

The results for accuracy are given in Table 2. While not all inputs produced are valid, almost all valid inputs derived are actually failing. The term "abstract failure-inducing input" thus is adequate. This is especially true for simple languages such as *find* and *grep* command lines, which perform very well in terms of accuracy.

Instantiating abstract failure-inducing inputs produced by DDSET *produces failures with high accuracy. On average, 86.5% of inputs produced were valid, of which 99.9% succeeded in reproducing the failure.*

11 THREATS TO VALIDITY

Our evaluation has the following threats to validity:

External Validity. The external validity (generalizability of our results) of our results depends on how representative our data set is. Our case study was conducted on a small set of programming language interpreters and UNIX utilities. Further, we evaluate only a limited number of bugs for their abstractability. Hence, there exist a threat that our samples may not be representative of the real world. A mitigating factor is that these were real bugs logged by real people, and the grammars we used are from some of the most popular and complex programming languages that are used to build real world applications. Hence, we believe that our approach can be generalized to other subjects, especially subjects with less complex languages.

Internal Validity. The threat to internal validity refers to the correctness of our implementation and evaluation. We mitigated this by verifying that all our algorithms and programs work on a small set of well understood examples before applying it to our sample set of bugs. Every resulting abstract failure-inducing input is simple enough to quickly spot errors.

Construct Validity. The threat to our construct validity (are we measuring what we claim to be measuring) is how useful the effectiveness and accuracy measures are. Short of a user study, this threat cannot be completely mitigated. However, we note that our abstraction intuitively models how a developer thinks about a fault causing input, i.e. "here is a variable, and if I use this particular variable in this fashion later, it triggers a bug". We also note that simplification and generalization seem to be qualities whose usefulness for users is universally accepted.

12 RELATED WORK

Despite their importance for debugging, the study of failure-inducing inputs and their characteristics is very limited. Notably, the question of determining *sets* of failure-inducing inputs has, to the best of our knowledge, never been addressed in the literature; DDSET thus opens the door to a new field of research.

Generalizing failure-inducing inputs has been mostly studied in the context of *reducing* them. The original algorithm for *delta debugging* for program reduction was introduced by Zeller and Hildebrandt [19]. Delta debugging works by partitioning the input sequence into chunks and checking whether smaller and smaller chunks can be discarded while retaining the required property.

A number of variants for delta debugging exist. The first research to target structured inputs was HDD [12] by Misherghi and Su. HDD was motivated by finding that delta debugging performs poorly on structured inputs. HDD applies delta debugging on a single level of the hierarchy at any given time. Herfert et al [10] introduced tree transformations for hierarchical reduction and showed that in typical programming languages, parent nodes could often be replaced by one of the child nodes, leading to better reduction. Perses [18] by Sun et al. uses an *input grammar* to guide reduction. The innovation of Perses is in realizing that one could use the node type and look for similar node types in the descendent nodes for replacement. The other innovation is in identifying that certain kinds of nodes can be entirely deleted if the nonterminal definition has an empty rule. DDSET uses Perses as its initial reduction step and generalizes from there, using the input grammar also to produce additional test inputs to validate abstractions.

Reduction algorithms can also be specialized for individual input languages. ChipperJ [17] uses Java specific transformations to produce a minimal cause preserving input. A similar research is C-Reduce by Regehr et al [16] for C programs. Similar to ChipperJ, it applies a sequence of valid reducing transformations on C code to obtain a minimal cause preserving input. ChipperJ and C-Reduce achieves minimization through their in depth semantic knowledge of the Java and C programs, and together outperform other more general variants of hierarchical delta debugging. The identification of context-sensitive nodes, as implemented in DDSET, specifically targets identifier definition and usage in programming languages.

Among the specializations of delta debugging for specific domains, Bruno's SIMP tool [5] for reducing failure-inducing SQL queries stands out in that it attempts to distinguish fixed (necessary) parts in the failure-inducing input from variable parts. In

contrast to DDSET, SIMP does not attempt to generalize inputs to a maximum or to produce a abstract failure-inducing input.

Another notable approach close to our own is *universal sub-value generalization* by Lee Pike [15]. The approach by Pike shows how one can use a *forall* construct in *Haskell* to indicate generalizable constructors in the *Haskell* representation of an AST. However, compared to our approach, Pike's approach (1) does not generate a compact representation, (2) does not understand how to deal with lexical elements, (3) produces *unsound* generalization when independent sub-causes exist, and (4) does not identify context-sensitive parts such as defined variables (sharing).

Perses and DDSET require a grammar for their reduction, abstraction, and testing steps. Such grammars would typically be specified manually, which can be quite some effort. Recent work on mining grammars from dynamic control flow [9] suggests that such grammars can also be extracted from programs by dynamically tracking individual input characters. In our setting, such a learner could be applied to produce a grammar from the failing program and its failure-inducing input, requiring no further specification effort.

13 CONCLUSION AND FUTURE WORK

What are the inputs that cause a failure? We present the first algorithm that not only reduces a given failure-inducing input to a short string, but also *generalizes* it to a set of related failure-inducing inputs. This abstract failure-inducing input is short and easy to communicate, and represents the set of failure-inducing inputs with high accuracy. By producing sets rather than single inputs, DDSET has a clear advantage over classical reduction algorithms.

In our future work, we want to take the generalizations pf DDSET further—notably by systematically exploring the surroundings of the original input, not only reducing it, but also by adding additional elements in order to determine the context under which a failure takes place or not. Eventually, we thus want to be able to determine an entire *language* that not only serves as a producer of failure-inducing inputs, but also as a *recognizer*, being able to predict whether a given input will cause a failure or not. We expect several usage scenarios for such a recognizer, notably as it comes to detecting malicious inputs.

DDSET and all evaluation data is available as open source at

https://github.com/vrthra/ddset

ACKNOWLEDGMENTS

Andreas Zeller's team at CISPA provided helpful comments on earlier revisions of this paper. We also thank the anonymous reviewers for their highly constructive comments.

REFERENCES

[1] Antlr. 2020. ANTLR Clojure Grammar. https://github.com/antlr/grammars-v4/tree/master/clojure. Online; accessed 27 January 2020.
[2] Antlr. 2020. ANTLR Javascript Grammar. https://github.com/antlr/grammars-v4/tree/master/javascript/javascript. Online; accessed 27 January 2020.
[3] Antlr. 2020. ANTLR Lua Grammar. https://github.com/antlr/grammars-v4/tree/master/lua. Online; accessed 27 January 2020.
[4] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2014)*. 105–115.
[5] Nicolas Bruno. 2010. Minimizing Database Repros Using Language Grammars. In *Proceedings of the 13th International Conference on Extending Database Technology* (Lausanne, Switzerland) *(EDBT '10)*. Association for Computing Machinery, New York, NY, USA, 382–0393. https://doi.org/10.1145/1739041.1739088
[6] Clojure. 2020. Clojure Bugs. https://clojure.atlassian.net/secure/Dashboard.jspa. Online; accessed 27 January 2020.
[7] Google. 2020. Closure 2842 Bugs. https://github.com/google/closure-compiler/issues/2842. Online; accessed 27 January 2020.
[8] Google. 2020. Closure Bugs. https://github.com/google/closure-compiler/issues. Online; accessed 27 January 2020.
[9] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*.
[10] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-structured Test Inputs. In *IEEE/ACM Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 861–871. http://dl.acm.org/citation.cfm?id=3155562.3155669
[11] Lua. 2020. Lua Bugs. https://www.lua.org/bugs.html. Online; accessed 27 January 2020.
[12] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *International Conference on Software Engineering*. 142–151.
[13] Mozilla. 2020. Rhino 385. https://github.com/mozilla/rhino/issues/385. Online; accessed 27 January 2020.
[14] Mozilla. 2020. Rhino Bugs. https://github.com/mozilla/rhino/issues/385. Online; accessed 27 January 2020.
[15] Lee Pike. 2014. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 53–64. https://doi.org/10.1145/2633357.2633365
[16] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.
[17] Chad D Sterling and Ronald A Olsson. 2007. Automated bug isolation via program chipping. *Software: Practice and Experience* 37, 10 (2007), 1061–1086.
[18] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided Program Reduction. In *International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236
[19] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (Feb 2002), 183–200. https://doi.org/10.1109/32.988498