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 (14) 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 (11), (3 \times 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

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 parentheses expressions? Any parenthesized expression? Double 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.

\[
1 + ((2 * 3 / 4)) \quad ((\expr))
\]

(a) Failure-inducing input \quad (b) Abstract failure-inducing input

Figure 1: Input for the calculator program.

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 \(\langle(\langle expr\rangle)\rangle\) 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 \(\langle(\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\langle\<}
inducing failures. This will lead to the erroneous conclusion that the final abstraction is \((\text{expr}) \ (\text{op}) \ (\text{expr}) \ (\text{op}) \ (\text{expr})\). Our isolation step will correctly extract \(((\text{expr})) \ (\text{op}) ((\text{expr})) \ (\text{op}) ((\text{expr}))\). 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 \(\langle \$\text{var1} \rangle \ (\text{op}) \ \langle \$\text{var1} \rangle\) precisely identifies \(\langle \text{var1} \rangle\) 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 single quotes are represented by a nonterminal element such as \((\text{BEGIN})\), 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, \(1 + ((2 \times 3) / 4)\), the symbols ”1” ”+” ”((” ”2” ”*” ”3” ”)” ”/” ”4” ”)” form our input.

**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, \(\langle \text{expr} \rangle\) 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^4\text{ derivation tree}^4.

**Rule.** A finite sequence of terminals and nonterminals that describe an expansion of a given nonterminal. For our example, \(\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \ (\text{op}) \ \langle \text{expr} \rangle\) is one of the rules that defines the nonterminal \(\langle \text{expr} \rangle\) 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 \(1 + ((2 \times 3) / 4)\) 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 \(2 \times 3\) is an \(\langle \text{expr} \rangle\). Similarly, the root node for the expression \(2 \times 3 / 4\) is also an \(\langle \text{expr} \rangle\). 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.

\[^2\]These are marked as \text{SKIP} in ANTLR grammars.

\[^4\]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.

\[^5\]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.
SGS 1, July 18–22, 2020, Virtual Event, USA

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, Andreas Zeller

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

(a) Failure-inducing input

var { (sld): (sld) => {}} = (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, θ))
    while p_q ≠ ∅
        dtree, path ⇐ p_q.pop()
        snode ⇐ dtree.get(path)
        trees ⇐ ∅
        compatible_nodes ⇐ snode.get_all_nodes(snode.key)
        if θ ∈ grammar[snode.key] then
            compatible_nodes.append(θ)
        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 (θ_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.
which successfully triggers the

At this point, we will have a minimal input where the predicate
generated string.
identify which parts of the given derivation tree are required to
abstracting the non-causative parts. For abstraction, the idea is to
contributed directly to the failure observed, and identifying and
is the process of identifying the causative parts that
Abstraction

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 re-
place 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 reproduc-
ing the failure. Collect every such alternative tree, and iden-
tify 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 cor-
respond 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.

(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 alterna-
tives 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 ab-
stracting ((start)$^0$). The generated strings for ((start)$^0$) contain strings
such as 1 + 0, 2 + 3 etc. These result in PASS from the predicate. That
is, ((start)$^0$) cannot be abstracted, and hence, marked as concrete. Simi-
larly, ((expr)$^1$), also cannot abstracted, and marked as concrete. Con-
sidering ((expr)$^2$), all generated strings produced are parenthesized.
However, only a few are double parenthesized. Hence, ((expr)$^2$) is
also marked as concrete. Considering ((expr)$^3$), all generated strings
produced are of the form //((...))// which successfully triggers the
predicate. Hence, the path to ((expr)$^3$) 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 ab-
stract derivation tree encodes what exactly caused the failure. In our
case, the abstract derivation tree generates the string ( ((<(expr)) ) ),
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 exam-
ple, here is a minimized input string [7] for the *closure* JavaScript
compiler: 
```
 while ((1_0)) { if ((1_0)) { break; ; var 1_0; continue } } }
```
The problem here is that one requires exactly two instances of 1_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 1_0, and succeed because
the other two instances are present in the string. Similarly, the
second and third 1_0 will also be identified as abstractable because
identify 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. Then we 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

<table>
<thead>
<tr>
<th>Algorithm 2 The abstraction and isolation algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function</strong> can_generalize(tval, dtree, grammar, predicate, rnodes) checks ⇐ 0 while checks &lt; 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</td>
</tr>
<tr>
<td><strong>function</strong> 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(dtree.children) do tval ⇐ (child, path, UNCHECKED) paths.append(abstraction(tval, dtree, predicate, rnodes)) end for return paths end if</td>
</tr>
<tr>
<td><strong>function</strong> isolation(tree, grammar, predicate) unv ⇐ [{∅, UNCHECKED}] verified ⇐ ∅ original ⇐ ∅ while unv ≠ ∅ do v ⇐ unv.shift() notpv ⇐ filter(original, λo :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)</td>
</tr>
</tbody>
</table>

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. Then we 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
Figure 7: A derivation tree for \texttt{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 \texttt{d} has a string representation \texttt{v=0} while \texttt{k, e, p,} and \texttt{q} has the same string representation \texttt{v}. We only collect string representations of concrete nodes (annotated by ‘\*’ in Fig. 7). From the figure, we can see that the nodes \texttt{a} and \texttt{b} have similar string representation — \texttt{v=0; v=1/v} similarly, \texttt{c} and \texttt{d} have \texttt{v=0}, and \texttt{e, k, p,} and \texttt{q} have \texttt{v} and \texttt{s} so on. Out of these, we eliminate nodes that are a child of any of the other nodes in the grouping. This eliminates \texttt{b} since it is a child of \texttt{a}, which also eliminates the grouping since there is only one member left in the group.

This leaves us with one group, with members \texttt{e, k,} and \texttt{p} corresponding to \texttt{v}. Next, we pick a randomly generated value for one of the nodes; Say \texttt{x}, and use this value in place of each node, generating a new input string \texttt{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 \texttt{myval="myval"; check(myval)}. The input produces a failure if a variable with value “\texttt{myval}” is passed to the function \texttt{check()}. Here, \texttt{myval} can be replaced by any legal variable name so long it is correctly passed to \texttt{check()}. However, the string value has to remain “\texttt{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 \texttt{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

\begin{verbatim}
function IDENTIFY_SHARED_NODES(tree, grammar, predicate)
cpaths ← concrete_paths(tree)
snodes ← find_similar(tree, cpaths)
tree ← tree
for key ∈ snodes do
    plst ← snodes[key]
    paths ← find_shared(plst, grammar, tree, predicate)
    if paths then
        mtree = mark_context_sensitive(paths, mtree)
    end if
end for
return mtree
end function

function CONCRETE_PATHS(tree, paths)
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(plet, grammar, tree, predicate)
for paths ∈ len_sorted_combinations(plet) 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 0
end function
\end{verbatim}
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 (\textit{Id}) to be the same identifier (baz in the minimized string) and the first (\textit{)} to be present.

Second, the abstract failure-inducing input suggests that the token (\textit{variableDeclaration}) is effectively a space filler, as any instantiation of (\textit{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 (\textit{Id}) and (\textit{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 \textit{lexical} and \textit{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 \textit{invisible} elements, and remove them from the compact representation of the abstract failure-inducing input.

Finally, tokens such as \textit{begin} and \textit{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 \textit{reduction} function, which minimizes the derivation tree to a 1-minimal-tree input. The 1-minimal-tree is then passed to the \textit{isolation} function. The isolation function in turn, uses abstraction to identify nodes that are independently abstract. Finally, the function \textit{identify_shared_nodes} identifies and marks parts that are shared. The tree thus produced is converted to a string representation and returned.

Algorithm 4 Top level

\begin{verbatim}
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
\end{verbatim}

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, \textit{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 \textit{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 \textit{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 \(O(n \log(n))\) where \(n\) is the number of tokens, and the base of the logarithm is the branching factor. Furthermore, \textit{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 syntactically 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** 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** 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 DBG Bench [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 DBG Bench.

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.

![Table 1: Results for abstraction](image)

- 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.

  - 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 bugs 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.

<table>
<thead>
<tr>
<th>Table 1: The effectiveness of abstraction on reported bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>lua-5.3.5.4</td>
</tr>
<tr>
<td>clj-2345</td>
</tr>
<tr>
<td>clj-2450</td>
</tr>
<tr>
<td>clj-2473</td>
</tr>
<tr>
<td>clj-2518</td>
</tr>
<tr>
<td>clj-2521</td>
</tr>
<tr>
<td>closure 1978</td>
</tr>
<tr>
<td>closure 2808</td>
</tr>
<tr>
<td>closure 2842</td>
</tr>
<tr>
<td>closure 2937</td>
</tr>
<tr>
<td>closure 3178</td>
</tr>
<tr>
<td>closure 3379</td>
</tr>
<tr>
<td>rhino 385</td>
</tr>
<tr>
<td>rhino 386</td>
</tr>
<tr>
<td>grep 3c3bace</td>
</tr>
<tr>
<td>grep 54d55bba</td>
</tr>
<tr>
<td>grep 9c45c193</td>
</tr>
<tr>
<td>find 07b941b1</td>
</tr>
<tr>
<td>find 93623752</td>
</tr>
<tr>
<td>find c8491c11</td>
</tr>
<tr>
<td>find dbcb10e9</td>
</tr>
</tbody>
</table>

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),...
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 descendend 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.
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 of 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