Pardis: Priority Aware Test Case Reduction

Golnaz Gharachorlu(✉) and Nick Sumner

Simon Fraser University, Burnaby, BC, Canada
{ggharach,wsumner}@sfu.ca

Abstract. Test cases play an important role in testing and debugging software. Smaller tests are easier to understand and use for these tasks. Given a test that demonstrates a bug, *test case reduction* finds a smaller variant of the test case that exhibits the same bug. Classically, one of the challenges for test case reduction is that the process is slow, often taking hours. For hierarchically structured inputs like source code, the state of the art is Perses, a recent grammar aware and queue driven approach for test case reduction. Perses traverses nodes in the abstract syntax tree (AST) of a program (test case) based on a priority order and tries to reduce them while preserving syntactic validity.

In this paper, we show that Perses' reduction strategy suffers from *priority inversion*, where significant time may be spent trying to perform reduction operations on lower priority portions of the AST. We show that this adversely affects the reduction speed. We propose Pardis, a technique for priority aware test case reduction that avoids priority inversion. We implemented Pardis and evaluated it on the same set of benchmarks used in the Perses evaluation. Our results indicate that compared to Perses, Pardis is able to reduce test cases 1.3x to 7.8x faster and with 46% to 80% fewer queries.

Keywords: Test case reduction · Automated debugging · Priority aware reduction

1 Introduction

Test case reduction is a technique that aids in testing and debugging software. When an input for a program causes the program to exhibit a property of interest, like a bug, finding a smaller input that also exhibits the property can help to explain the behavior [1–3]. Given an input $I \in \mathbb{I}$ and an oracle $\psi : \mathbb{I} \rightarrow \mathbb{B}$ that performs a test and returns true iff a property holds, test case reduction aims to find a smaller input I' such that $\psi(I') = \text{true}$. Often, this problem is approached through Delta Debugging (DD), a longstanding and effective algorithm for test case reduction that essentially generalizes binary search [2]. However, for inputs with significant structure, generic DD can perform poorly, requiring significant time and not performing much reduction [3,4]. For compilers in particular, where the inputs must be valid programs, this has led to specialized techniques like Hierarchical Delta Debugging [3,4], language specific reducers like C-Reduce [5], and most recently to Syntax Guided Program Reduction as seen in Perses [6].

Syntax Guided Program Reduction (SGPR) is the present state of the art for compiler targeted test case reduction. The intuition behind SGPR is that the grammar defining the language of inputs eliminates many invalid sub-inputs from the search space. For example, when an input must adhere to the C programming language [7], removing the return type of a function declaration would not be valid because the C grammar specifies that the return type is required. Such syntactically invalid inputs are removed from the search space by SGPR.

Perses, a form of SGPR, takes as arguments not only a program p and oracle ψ, but also the context free grammar G of valid inputs [6]. It transforms the grammar so that removable parts of the input can be identified by the names of the grammar rules used to parse them. This also normalizes the grammar so that all removable components are expressed through quantifiers in an extended context free grammar [8], i.e. optionality (`?`) and lists (`*`, `+`). This transformation is illustrated in Fig. 1. Notice, for instance, that the recursive rule BAR denoting a list is transformed ($\Longrightarrow$) into a Kleene-+ quantified list. Individual elements of the list may be removed while preserving syntactic validity. Perses then parses the input of interest into an abstract syntax tree (AST) and traverses the AST while trying to (1) remove optional nodes and (2) perform DD to minimize the children of nodes representing lists. The grammar transformations have the benefit of making many syntactically correct removals easy and efficient to locate.

$$\mathsf{FOO} \rightarrow \mathsf{a} \mid \mathsf{a\ b} \Longrightarrow \begin{array}{l}\mathsf{FOO} \rightarrow \mathsf{a\ FOO\_opt}\\ \mathsf{FOO\_opt} \rightarrow \mathsf{b?}\end{array}$$

(a) Optional elements like b are refactored into rules with `?` quantifiers.

$$\mathsf{BAR} \rightarrow \mathsf{c} \mid \mathsf{c\ BAR} \Longrightarrow \begin{array}{l}\mathsf{BAR} \rightarrow \mathsf{BAR\_plus}\\ \mathsf{BAR\_plus} \rightarrow \mathsf{c+}\end{array}$$

(b) Lists of elements are refactored into rules with `*` or `+` quantifiers.

Fig. 1. Overview of Perses grammar transformations for SGPR.

Perses has significantly improved the speed of program reduction. However, it still takes several hours to reduce some inputs. Consider the code in Listing 1.1 along with its AST in Fig. 3. This example is similar to a C program generated by the compiler testing tool CSmith [9]. In this example, Perses first considers the root node with ID $\textcircled{1}$ of the AST. Since the rule for this node ends in _star, it is a list node, and its children are the elements of the list. Thus, Perses applies DD to the list of children for node $\textcircled{1}$ to minimize the number of children. When such lists are long, significant time can be devoted to this task. We show in Sect. 4 that this can lead to substantial *stalls* in reduction, where no progress is made while a list is being processed. However, most of the children of this node have low *token weight*, the number of tokens beneath a given node that is denoted by `w:` in Fig. 3. Indeed, greater value would be found by focusing on just *one* of its children, node ⑤, which contains the majority of the input beneath it. By spending greater effort up front on portions of the AST of lesser value, Perses suffers from a form of *priority inversion.* Priority inversion occurs when a low priority task is scheduled instead of a high priority task. In this case, Perses focuses on removing low token weight nodes instead of high token weight nodes. Indeed, Perses may even fail to remove elements that would enable better reduction success overall. In this case, the declarations of `foo`, `S`, and `d` are used within the code beneath node ⑤. Thus, those uses need to be eliminated *before* any of the declarations can be removed successfully. In practice, we find that priority inversion has a significant impact on reduction time in SGPR.

To address priority inversion, we have developed *priority aware reduction strategies* for program reduction. By focusing the reduction effort on the nodes of the AST that cover the greatest number of tokens, we prioritize reduction of the most complex parts of the input first. This has multiple important benefits: (1) Dependencies between program elements are more likely to be broken by eliminating the complex uses first. (2) Stalls in reduction from unsuccessful rounds of DD can be mitigated. (3) By removing large portions of an input earlier on, each oracle query to ψ can take less time because smaller inputs tend to be faster to check. We have designed and evaluated a tool, Pardis, that makes use of these techniques and found that it leads to consistent and significant performance improvements over Perses on the Perses benchmarks [6].

In summary, this paper makes the following contributions:

1. **Priority awareness.** We identify *priority inversion* as a key problem facing SGPR techniques and develop priority aware reduction strategies as a potential solution. *Priority aware reduction strategies* focus the reduction effort on the complex portions of an input first, enabling earlier and thus faster test case reduction (Sects. 3, 4.1).
2. **Optimization.** We identify redundancies in the reduction process when using Perses' transformed grammars and develop a solution to prune them from the candidate search space (Sect. 3.2).
3. **Significant performance improvement.** We implemented our strategies in a tool, Pardis, and evaluated it on the same benchmarks used by Perses. Experimental results show that Pardis both removes more of the input earlier on and is faster overall. Compared to Perses, Pardis reduces test cases 1.3x to 7.8x faster and with 46% to 80% fewer oracle queries (Sect. 4.1).

2 Background and Motivation

Consider again the example in Fig. 3 and suppose that the oracle (ψ) checks that this program p should print `"Hello World!"` on line 24 (marked with $*$). Thus, the smallest subprogram for which ψ returns true is the main function with the desired `print` statement.

To search for this smaller input inside the original input, Perses traverses the AST using a priority queue ordered by the token weight. In each trial, the node

Listing 1.1: A C program with property of interest on line 24.

```
1  double d = 0.10;
2  struct S {
3    int f1;
4    int f2;
5  };
6  void foo(struct S s, char str[]){
7    double v = s.f2 + s.f2 * d;
8    printf("%s %f\n",str,v);
9  }
10 int main() {
11   unsigned int a = 1;
12   char b[] = "first";
13   char c[] = "second";
14   if (a) {
15     struct S s1;
16     s1.f1 = 1;
17     s1.f2 = 4000;
18     struct S s2;
19     s2.f1 = 2;
20     s2.f2 = 2000;
21     foo(s1, b);
22     foo(s2, c);
23   }
24   printf("Hello World!\n"); (*)
25   return 0;
26 }
```

node(s) to remove	removed
{2,3,4,5}	F
{2,3}	F
{4,5}	F
{2}	F
{3}	F
{4}	F
{5}	F
{3,4,5}	F
{2,4,5}	F
{2,3,4}	F
{2,3,5}	F
{8,9,10,11,12,13}	F
{8,9,10}	F
{11,12,13}	F
{8,9}	F
{10,11}	T
{12,13}	F
{8}	T
{9}	T
{12}	F

(a) Perses

node to remove	removed
{1}	F
{5}	F
{7}	F
{11}	T
{4}	T
{3}	T
{10}	T
{9}	T
{8}	T
{12}	F
{2}	T

(b) Pardis

node(s) to remove	removed
{1}	F
{5}	F
{7}	F
{11}	T
{4}	T
{3}	T
{9,10}	T
{8}	T
{12}	F
{2}	T

(c) Pardis Hybrid

Fig. 2. One round of removal trials in Perses, Pardis and Pardis Hybrid for the AST in Fig. 3. Numbers are node IDs.

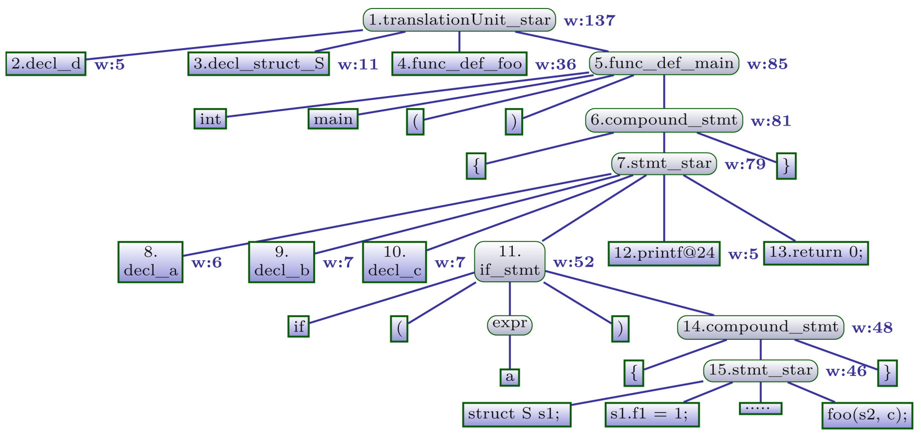


Fig. 3. AST of the program in Listing 1.1. `w` denotes the token weight of each node.

with the maximum weight is removed from the work queue and traversed. In our example, the queue starts out containing only the root of the AST, node ①. Perses performs specific reduction operations on different types of nodes during traversal. For instance, on optional nodes, Perses tries to remove the optional child node. For list nodes, Perses minimizes the list of children using DD. Any *remaining* children of the traversed node are then added to the priority queue in order to be traversed in the future.

Observe that in this example, Perses will first examine node ① and remove it from the queue. Because ① is a list node, DD is applied to the children of ①. Different combinations of children are removed from ① and the result is checked by ψ to find a smaller input. First, all children are removed and ψ is checked. After this fails, the first half of the children (② and ③) are removed, but ψ returns false again because this removes required declarations. Since removing the second half of the children (④ and ⑤) also fails, the process continues recursively. First DD tries shrinking the list by *removing* each individual child, and next it tries *only keeping* each individual child. Ultimately none of the trials succeed, so all children are added to the queue, and reduction continues with node ⑤. The intervening node ⑥ is not tested by SGPR because it is not syntactically removable. The next node removed from the work queue is node ⑦. This continues until the queue is empty. The precise trials exercised in this process are illustrated in Fig. 2(a). Note that 16 steps elapse until a successful trial occurs.

While the priorities used by Perses are controlled by the token weight, they determine how the *children* of the traversed nodes are removed. Thus, any node whose *parent* in the AST is a list is given the same priority as all other elements in the list. This is because DD recursively tries to minimize the entire list until no single element can be removed, regardless of the priorities of individual list elements. As a result, Perses must employ DD on the entirety of the children of ① even though it would be more beneficial to focus on just one child, node ⑤.

Instead, Pardis more directly models the priorities. We note that in an optional or list node, such as ①, each child may be removed in a syntactically valid fashion. We call such removable nodes *nullable*. When traversing a nullable node in the AST, we can simply try directly to remove it, adding its children if the removal fails. For instance, in the running example, we would visit ① first. Because ① cannot get removed, we would simply add its children to the priority queue. Note that all children of ① are nullable, but ⑤ has the highest *token weight*. Thus, we next select ⑤ to traverse but removing ⑤ also fails. From the given token weights, we next traverse ⑥, which is syntactically not removable, and then ⑦, which we attempt to remove but is unsuccessful. Next ⑪ is visited and successfully removed. Removing ⑪ *enables the removal of* ④, ③ *and* ②. Thus, they are removed in a single pass of the tree using Pardis, whereas Perses would require multiple traversals of the AST to remove them. This process continues until the desired output is achieved. As seen in Fig. 2(b), just 4 steps elapse until the first successful trial removes node ⑪.

Note that in this example, Pardis is able to reduce to the desired output in a *single pass*, while Perses requires multiple passes of the AST. In practice, all program reduction techniques continue until a fixed point is reached, including Pardis, however Pardis can achieve greater reduction in a single traversal of the AST, accelerating convergence on the fixed point.

This priority aware approach can still have drawbacks, however. After focusing on the highest priority nodes, there may be many lower priority nodes remaining. For example, there are multiple remaining nodes of weight 7 in the tree after performing the reduction by Pardis as described above. We also show experimentally that these lower priority nodes occur in practice in Sect. 5. The above approach of Pardis considers each node *one at a time*, which can have poor performance when reducing such long lists. In addition, we thus propose a *hybrid* approach that still prioritizes nodes by maximum token weight but also uses a list based reduction technique for spans of nodes that have *the same* token weight. This hybrid approach is able to achieve the benefits of being priority aware while still avoiding the cost of considering each node of the AST individually.

Section 3 presents the algorithms behind these techniques in detail.

3 Approach

Recall that the core of Pardis, similar to Perses, maintains a priority queue of the nodes in an AST and traverses the nodes in order to process them. It also makes use of Perses Normal Form, the result of the grammar transformations that Perses introduced [6]. The key difference is that instead of using the token weight of a parent node to determine when its nullable children may be removed, Pardis identifies all nullable nodes (see Sect. 3.2) and uses their token weights directly to prioritize the search. The core algorithm for this process is quite straightforward and presented in Algorithm 1.

Algorithm 1: Priority queue driven program reduction.

```
Input: $P : \mathbb{P}$ – The program to reduce as an AST
Input: $\psi : \mathbb{P} \to \mathbb{B}$ – Oracle for the property to preserve
Input: $\rho : \mathbb{V} \to \mathbb{N} \times \cdots \times \mathbb{N}$ – Prioritizer for AST nodes
Result: A minimum program $p \in \mathbb{P}$ s.t. $\psi(p)$
1 work ← MaxPriorityQueue({p.root}, $\rho$)
2 while $!work.empty()$ do
3     node ← work.takeMax()
4     if $node.isNullable$ && $\psi(p - node)$ then
5         p ← p - node
6     else
7         work.insert(node.children)
8 return $p$
```

Line 1 of the algorithm constructs the priority queue (a max-heap), initializing it with the root of the AST and using a parameterizable priority ρ. ρ is simply a function that takes a node and returns its priority as a tuple. The priority queue selects the element with a lexicographically maximal priority, so ties on the *first* element of the priority tuple are broken by the *second* element and so on. As seen in Fig. 4, for Pardis, ρ_{Pardis} returns a pair of numbers, the token weight of the node and the position of the node in a decreasing, right-to-left, breadth first search. The specific breadth first order means that for an AST with n nodes, bfsOrder(p.root)=n, the last child c of p.root has bfsOrder(c)=n-1, and so on. Thus, if several nodes have the same token weight, the one highest in the AST and furthest to the right is selected next. This ordering decreases the chances of trying to remove a declaration before its uses [10].

Line 2 starts the core of the algorithm. While there are more nodes to explore in the queue, the node with the next highest priority is considered. If it is nullable and can be successfully removed, we remove it from the AST, otherwise we add its children to the queue so that they will also be traversed.

While the algorithm is surprisingly simple, we have found it to perform significantly better than the state of the art in practice. As we explore in Sect. 4.2, this results from prioritizing the search toward those portions of the input where reduction can have the greatest impact. To more closely compare with Perses, consider a version of Perses that upon visiting a list or optional node only tries removing each child of that node once[1]. This "one node at a time" variant of Perses can also be implemented using Algorithm 1 by carefully choosing the priority formula ρ. Because Perses considers removing the *children* of the nodes it traverses, it actually prioritizes the work queue using the token weight of the *parent* rather than the token weight of nullable nodes being considered for removal. This leads to the alternative prioritizer ρ_{perses} presented in Fig. 4. Observe that all children of a list node receive the same token weight, that of the entire list. This can inflate the priority of some nodes in the work queue and leads to poor performance.

Definitions:

tokensBelow(n) – returns the number of tokens beneath an AST node n.

bfsOrder(n) – returns the position of an AST node n in a decreasing, right to left, breadth first search.

Prioritizers:

$$\rho_{\text{PARDIS}} = (\mathsf{tokensBelow(n)},\ \mathsf{bfsOrder(n)})$$

$$\rho_{perses} = \begin{cases} \textbf{let}\ \mathsf{parentWeight} \leftarrow \mathsf{tokensBelow(n.parent)}\ \textbf{if}\ \mathsf{n.parent}\ \textbf{else}\ \infty\ \textbf{in} \\ \textbf{let}\ \mathsf{parentOrder} \leftarrow \mathsf{bfsOrder(n.parent)}\ \textbf{if}\ \mathsf{n.parent}\ \textbf{else}\ \infty\ \textbf{in} \\ (\mathsf{parentWeight},\ \mathsf{parentOrder}) \end{cases}$$

$$\rho_{\text{PARDIS HYBRID}} = \begin{cases} \textbf{let}\ \mathsf{parentOrder} \leftarrow \mathsf{bfsOrder(n.parent)}\ \textbf{if}\ \mathsf{n.parent}\ \textbf{else}\ \infty\ \textbf{in} \\ (\mathsf{tokensBelow(n)},\ \mathsf{parentOrder},\ \mathsf{bfsOrder(n)}) \end{cases}$$

Fig. 4. Prioritizers used for PARDIS, node at a time Perses, and PARDIS HYBRID.

Like other program reduction algorithms [3,5,6,11,12], Algorithm 1 is used to compute a fixed point. That is, in practice the algorithm is repeated until no further reductions can be made. As in prior work, we omit this from our presentation for clarity. In theory, this means that the worst case complexity of the technique is $O(n^2)$ where n is the number of nodes in the AST. This arises when only one leaf of the AST is removed in each pass through the algorithm. In practice, most nodes are not syntactically nullable, and we show in Sect. 4.1 that performance of PARDIS exceeds the state of the art.

In addition, while we focus on *removing* nodes of the AST, Perses also tries to *replace* non-list and -optional nodes with compatible nodes in their subtrees. We do not focus on this aspect of the algorithm. In practice, we found it to

[1] We compare against *both* versions of Perses in Sect. 4.1.

significantly hurt performance (see Sect. 4.1) and we consider efficient replacement strategies to be orthogonal to and outside the scope of this work.

3.1 Pardis Hybrid

The initial priority aware technique from Algorithm 1 can also encounter performance bottlenecks, however. The original motivation for using DD on lists of children in the AST was that its best case behavior is $O(log(n))$ where n is the number of children in the list. This is because it tries removing multiple children at the same time. Processing one node at a time, however, requires that every list element is considered individually, guaranteeing $O(n)$ time for one round of Algorithm 1. Priority aware reduction that proceeds one node at a time faces a different set of inefficiencies that can still cause stalls in the reduction process.

Thus, we desire a means of removing multiple elements from lists at the same time while *still* preserving priority awareness. In order to achieve this, we developed Pardis Hybrid, as presented in Algorithm 2. This approach uses a modified prioritizer as presented in Fig. 4 that first orders by token weight, then by parent traversal order, then by node traversal order. The effect this has is that all children of the same parent with the same weight are grouped together. As a result, we can remove them from the priority queue together and perform list based reduction (like DD) to more efficiently remove groups of elements in a list that have the same priority (for instance, nodes ⑨ and ⑩ get removed as a group in one trial using Pardis Hybrid as shown in Fig. 2(c)). Because the search is still primarily directed by the token weights of the removed nodes, the technique still fully respects the priorities of the removed nodes.

Algorithm 2: Pardis Hybrid algorithm with priority aware list reduction.

```
    Input: $p : \mathbb{P}$ – The program to reduce as an AST
    Input: $\psi : \mathbb{P} \to \mathbb{B}$ – Oracle for the property to preserve
    Result: A minimum program $p \in \mathbb{P}$ s.t. $\psi(p)$
1   work ← MaxPriorityQueue({p.root}, $\rho_{\text{Pardis Hybrid}}$)
2   while !work.empty() do
3       nodes ← work.takeWithSameWeightAndParent()
4       nullable, nonnullable ← partitionNullable(nodes)
5       removed, retained ← minimize(p, nullable, $\psi$)
6       p ← p - removed
7       work.insert($\bigcup_{x \in \text{retained} \cup \text{nonnullable}}$ x.children)
8   return $p$
```

Similar to the previous approach, line 1 of Algorithm 2 starts by creating the priority queue. Note that it specifically uses the prioritizer $\rho_{\text{Pardis Hybrid}}$, which groups children having the same token weight in the priority queue. As long as there are more nodes to consider, line 3 takes all nodes from the queue with the same weight and parent. If the weight of a node is unique, this simply returns a list of length 1. Line 4 filters out non-nullable nodes from the trial, and line 5 just applies list based reduction to any nullable nodes. Lines 6 and 7 then remove the eliminated nodes from the tree and add the children of remaining nodes to the work queue. Again, this algorithm actually runs to a fixed point.

While the worst case behavior of DD is $O(n^2)$ [2], this can be improved to $O(n)$ by giving up hard *guarantees* on minimality [13]. Since this reduction process is performed to a fixed point anyway, minimize on line 5 makes use of this $O(n)$ approach to list based reduction (OPDD) without losing 1-minimality. As a result, the theoretical complexity of Pardis Hybrid is the same as Pardis.

3.2 Nullability Pruning

Finally, we observed that many oracle queries were simply unnecessary. Specifically, recall that a node can be tagged nullable because it is an element of a list or a child of an optional node, as previously defined by Perses grammar transformations [6]. The complete algorithm for this tagging is in *TagNullable* of Algorithm 3. However, for example, a list of one element could contain another list of one element. In the AST, this appears as a chain of nodes, at least two of which are nullable. Removing *any one* of these nodes removes the same tokens from the AST. Thus, it is only necessary to select a single nullable node from any *chain* of nodes, and the others can be disregarded.

We exploit this through an optimization called *nullability pruning*. We traverse every chain of nodes in the AST, preserving the nullability of the highest node in the chain and removing nullability from those below it. The complete algorithm is presented in *PruneNullable* of Algorithm 3. In effect, it is just a depth first search that removes redundant nullability from nodes along the way instantaneously.

In practice, we find that this can statically (ahead of time) prune most of the AST from the search space. Specifically, in the benchmarks we examine in Sect. 4, we find that of 1,593,875 total nullable nodes, 17% are redundant optional nodes and 44% are redundant list element nodes. We observe the impact of this pruning on the actual reduction process in Sect. 4.1.

Algorithm 3: Nullability tagging and pruning.

```
1  Function TagNullable(p)
       Input: $p : \mathbb{P}$ – The program to reduce as an AST
2      foreach Node $n \in p$ do
3          if $n \in KleeneStar \cup KleenePlus \cup Optional$ then
4              foreach $c \in n.children$ do c.isNullable ← true

5  Function PruneNullable(p)
       Input: $p : \mathbb{P}$ – The program to reduce as an AST
6      Function OptimizeBelow(n)
7          hasNullable ← false
8          Loop
9              if hasNullable then
10                 n.isNullable ← false
11             else if n.isNullable then
12                 hasNullable ← true
13             if $1 == |n.children|$ then
14                 break
15             n ← n.getOnlyChild()
16         foreach $c \in n.children$ do OptimizeBelow(n)
17     OptimizeBelow(p.root)
```

4 Evaluation

We evaluate Pardis's performance and examine the impact of priority inversion on reduction by answering the following research questions:

- **RQ1.** How does Pardis perform compared to Perses in terms of reduction time and speed, number of oracle queries, and size of the reduced test case?
- **RQ2.** Does priority inversion adversely affect the reduction efficiency? In particular, does reduction require more work with a traversal order suffering from priority inversion?

4.1 RQ1. Performance: Pardis vs. Perses

Experimental Set-Up. We evaluate Pardis on the set of C test cases used in the evaluation of Perses, including the oracle scripts provided by authors of Perses. While using these, we observed that they still allowed for some undefined behavior [5,14], so we updated all oracles to reject test case variants with undefined behavior. As a result, we were able to reproduce bugs for 14 out of 20 original test cases. The remaining benchmarks that could not reproduce their original failures were elided for this study. Since the implementation of Perses' components is not publicly available, we implemented the Perses grammar transformations and reduction based on the algorithms available in the paper [6] using the C++ bindings of ANTLR [15]. All of our implementations have been made available[2]. Our experiments were conducted on an Intel Xeon E5-2630 CPU and 64 GB memory running Ubuntu.

Variants of Reduction Techniques. To better explain performance differences, we benchmark several algorithms that each add one difference. All approaches compute fixed points as previously described.

- *Perses DD*- The removal-based algorithm of Perses that applies DD on children of list nodes [6].
- *Perses OPDD*- The same as Perses DD but using the $O(n)$ reduction algorithm of OPDD [13]. It is faster than Perses DD in practice.
- *Perses N*- The one node at a time Perses that does not apply DD on list elements but removes them one by one using Perses' parent oriented priorities.
- Pardis w/o Pruning- This uses the Pardis algorithm but does not apply nullability pruning optimization proposed in Sect. 3.2.
- Pardis- Our proposed removal algorithm that also applies nullability pruning.
- Pardis Hybrid- The hybrid version of Pardis with nullability pruning and OPDD as its version of DD.

[2] https://github.com/golnazgh/PARDIS.

Table 1. Original and reduced test case size and number of oracle queries.

Bug	$O(\#)$	Perses DD		Perses OPDD		Perses N		PARDIS w/o PRUNING		PARDIS		PARDIS HYBRID	
		$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$	$R(\#)$	$Q(\#)$
clang-22382	21,068	597	5,323	597	4,865	354	3,203	354	2,702	354	2,011	354	2,319
clang-22704	184,444	250	4,181	250	3,775	220	5,083	236	4,956	236	4,342	236	2253
clang-23309	38,647	1,624	8,688	1,624	8,095	1,522	6,106	1,726	4,618	1,726	3,004	1,726	3,684
clang-25900	78,960	618	4,455	618	4,020	600	2,816	618	2,343	618	1,652	618	1,997
clang-27137	174,538	725	9,035	725	8,299	681	6,858	807	5,889	807	4,293	807	4,891
clang-27747	173,840	379	3,171	379	2,845	311	1,773	313	1,418	313	1,074	308	1,218
clang-31259	48,799	821	4,457	821	4,073	821	3,282	538	2,464	538	1,662	538	1,853
gcc-64990	148,931	776	5,913	776	5,438	1,215	5,165	776	3,781	776	2,632	776	3,148
gcc-65383	43,942	462	5,503	462	5,002	486	3,502	598	2,559	598	1,839	598	2,204
gcc-66186	47,481	1,176	6,101	1,176	5,727	1,178	4,532	1,176	3,944	1,176	2,562	1,176	3,167
gcc-66375	65,488	1,232	7,989	1,232	6,780	1,198	4,202	1,232	4,512	1,232	3,036	1,232	3,851
gcc-70127	154,816	600	5,610	600	5,201	593	3,700	600	3,063	600	2,240	600	2,723
gcc-70586	212,259	1,583	7,671	1,583	7,276	1,489	5,582	1,497	5,233	1,497	3,491	1,497	4,318
gcc-71626	6,133	58	1,151	58	1,135	58	1,013	58	330	58	264	58	228
geomean	70300	609	5126	609	4705	583	3670	574	2881	574	2066	574	2270
median	72,224	672	5,556	672	5,102	640	3,951	609	3,422	609	2,401	609	2,521

O, R and Q denote number of tokens in the original test case, reduced one and total number of oracle queries performed by the reduction technique, respectively.

Reduction Performance. We compare these techniques in terms of *the number of oracle queries* (Q), *reduction quality* or size of the final reduced test case (R), *reduction time* (T), and *reduction speed* or the average number of tokens removed per second (E). Results are presented in Tables 1 and 2. The best values of queries, time, and speed are highlighted for each test case. As can be seen, in all cases, either PARDIS or PARDIS HYBRID outperform all variants of Perses. Compared to the full removal-based Perses algorithm (Perses DD), our proposed algorithms reduce **1.3x** to **7.8x** faster and with **46%** to **80%** fewer queries. The results across variants suggest that these benefits arise from priority awareness and nullability pruning. Due to fixed point computation, all approaches produce test

Table 2. Reduction time and speed for different variants of reduction techniques.

Bug	Perses DD		Perses OPDD		Perses N		PARDIS w/o PRUNING		PARDIS		PARDIS HYBRID	
	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$	$T(s)$	$E(\#/s)$
clang-22382	3,198	6	3,122	7	3,489	6	3,057	7	2,977	7	2,094	10
clang-22704	1,527	121	1,304	141	5,243	35	3,323	55	3,219	57	1,160	159
clang-23309	2,571	14	2,414	15	1,920	19	1,423	26	1,007	37	1,062	35
clang-25900	1,375	57	1,220	64	1,025	76	690	114	526	149	518	151
clang-27137	6,972	25	6,379	27	5,717	30	4,428	39	3,423	51	3,538	49
clang-27747	1,194	145	1,060	164	771	225	571	304	463	375	453	383
clang-31259	1,698	28	1,577	30	1,471	33	1,239	39	814	59	800	60
gcc-64990	1,980	75	1,768	84	1,981	75	1,237	120	932	159	916	162
gcc-65383	1,762	25	1,615	27	1,304	33	892	49	704	62	699	62
gcc-66186	1,583	29	1,493	31	1,299	36	1,016	46	691	67	741	62
gcc-66375	2,782	23	2,568	25	1,851	35	1,705	38	1,173	55	1,311	49
gcc-70127	3,083	50	2,812	55	2,265	68	1,520	101	1,124	137	1,173	131
gcc-70586	4,417	48	4,119	51	3,450	61	2,545	83	1,791	118	1,984	106
gcc-71626	156	39	156	39	206	29	57	107	54	112	20	304
geomean	1900	36	1750	40	1740	40	1202	58	933	75	807	86
median	1,871	34	1,692	35	1,886	35	1,331	52	970	64	989	84

T is reduction time in seconds. E is the efficiency of removal (number of tokens removed per second).

cases from which no one token can be removed while satisfying ψ (1-minimal) [2], but they can produce different final reduced test cases [2]. On average, Pardis yields reduced test cases with 574 tokens compared to Perses DD with 609 tokens.

In addition, we graphed the reduction progress of each test case for the different variants. Fig. 5 shows the percentage of remaining tokens over time during reduction. For sake of space, we only include graphs for six of the test cases. Note that the y-axis is log scaled. Pardis and Pardis Hybrid show much faster convergence to a reduced test case compared to Perses variants. Recall that the only factor differentiating Perses N from Pardis w/o Pruning is the order in which the queue of nodes is traversed. Unlike Perses N, Pardis w/o Pruning does not suffer from priority inversion and guides the reduction process based on token weights of the nodes to remove. As can be seen, this advantage leads to faster convergence to a reduced test case. We examine the impact of priority inversion on reduction speed more rigorously in Sect. 4.2.

Replacement. As mentioned in Sect. 3, Perses also considers a replacement strategy for non-list or -optional nodes in addition to removal for other nodes. For instance, in Fig. 3, Perses will attempt to replace node ⑥ with node ⑭ because they both match the same grammar rule (`compound_stmt`). This replacement fails since required declarations will get removed and ψ will return false.

Including replacement significantly increases the work done by reduction. For completeness, we implemented Perses DD with replacement as described in their paper [6] and defined a four-hour timeout for the reduction process. In 11 out of 14 cases, Perses DD with replacement could not finish the reduction process before reaching the timeout. In the remaining three, it generated reduced test cases with the same size or slightly smaller while performing a significantly larger number of oracle queries (more than 3$\times$ over Perses DD without replacement).

4.2 RQ2. The Impact of Priority Inversion

As shown in Fig. 5, avoiding priority inversion leads to faster convergence. One explanation for this is that priority awareness may decrease the amount of work required to remove a token (as seen in the motivating example). We explore this in a case study on `gcc-64990` with 148,931 tokens. The *number of removal attempts* for a token is number of times a single token is considered for removal. Removing any ancestor of a token in the AST will remove that token, so if a first attempt fails, a deeper ancestor may be attempted. We compute this for every token of the test case to get a sense of the work required for each token. A better traversal order of the AST should cause fewer overall token removal attempts. To measure only the impact of different traversal orders, we compare Pardis w/o Pruning with Perses N. As described in Sect. 4.1, they follow the exact same reduction rules and differ only in their traversal orders.

Figure 6 depicts histograms of the distributions of token removal attempts for Pardis w/o Pruning and Perses N. For clearer visualization, we show only the distributions for the number of attempts less than or equal to 20. We can see how Perses N distribution is inclined toward a larger number of removal attempts,

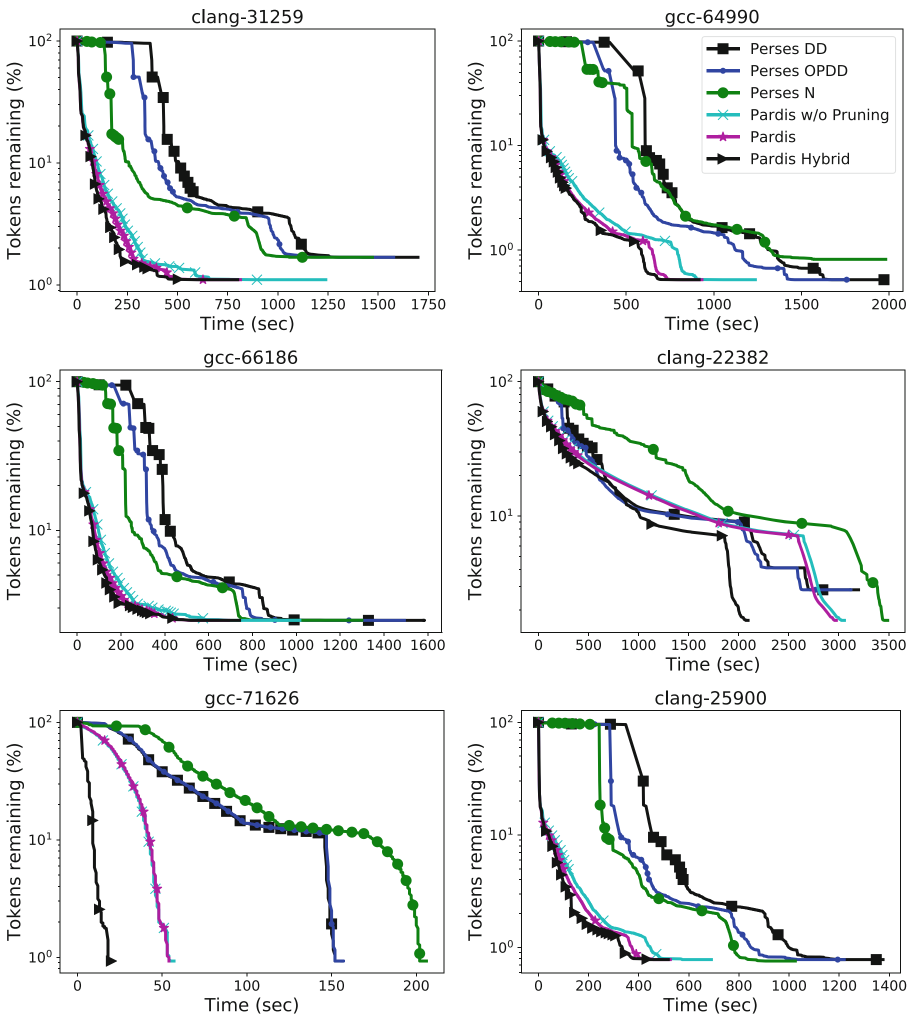


Fig. 5. Converging to a reduced test case in six variants of reduction techniques.

an indicator of more work required in order to remove individual tokens. In addition, we statically measure that the difference between the removal attempt distributions is significant. We use a one sided Wilcoxon rank-sum test [16] to determine whether the distribution of Perses N is indeed greater than that of PARDIS W/O PRUNING. The p-value computed for our data was less than $2.2e^{-16}$ which strongly supports this observation.

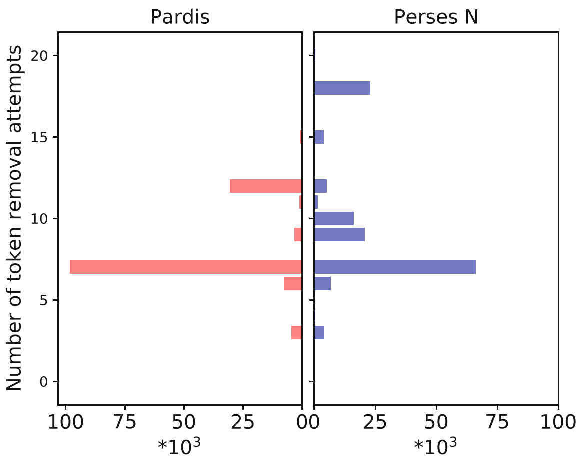


Fig. 6. Distributions of token removal attempts for Pardis w/o Pruning and Perses N.

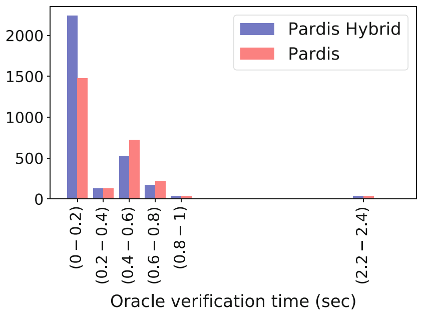
5 Discussion

Pardis Hybrid **as a *sweet spot* in reducing test cases:** As discussed earlier, unlike Perses, Pardis Hybrid does not suffer from priority inversion because it prioritizes the search primarily on the token weight of nodes being considered for removal. Moreover, unlike Pardis, it does not strictly remove one node at a time and allows the removal of nodes with the same weight and the same parent as a group. Hence, it can be considered a sweet spot in reducing test cases. We conduct two studies that can further explore this idea.

(1) Oracle Verification Time. The number of oracle queries is a common metric used in similar studies to reason about reduction efficiency since it directly impacts the total reduction time [2,3,6,13,17]. For instance, both Pardis and Pardis Hybrid perform fewer oracle queries and take less time than Perses. However, the number of oracle queries is not the only factor involved. The time required to run each of these queries, or *oracle verification time*, also affects the total running time. For instance, as presented in Sect. 4.1, Pardis has the smallest number of oracle queries in 12 out of 14 test cases. However, in terms of total reduction time and speed, Pardis Hybrid is the fastest in 8 out of 14 cases, even while performing *more* queries compared to Pardis in 6 of them. Oracle verification time can depend on multiple elements such as the size and complexity of the test case. Since Pardis Hybrid takes advantage of the possibility to remove more than one node at a time, it may try variants of the test case that are smaller and may be faster to verify compared to Pardis. To check this hypothesis, we conducted a case study on `gcc-64990` and recorded the running time of each oracle query during reduction. As shown in Tables 1 and 2, Pardis reduces this test case in 932 s with 2,632 queries, and Pardis Hybrid

has a total reduction time of 916 s (16 s shorter) while performing 3,148 oracle queries (516 more queries). Both techniques yield the same final test case.

Figure 7 depicts the distribution of oracle verification times in PARDIS and PARDIS HYBRID, showing that PARDIS has more queries that take longer compared to PARDIS HYBRID. The shorter queries in PARDIS HYBRID directly decrease its overall reduction time making it reduce test cases with fewer queries compared to Perses and shorter queries compared to PARDIS.



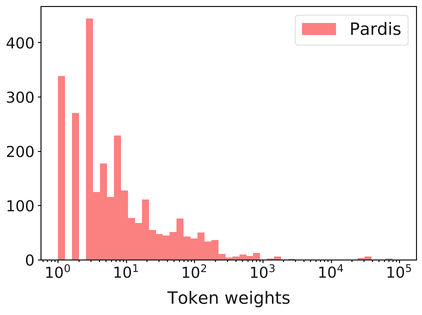


Fig. 7. Distribution of oracle verification time for PARDIS and PARDIS HYBRID.

Fig. 8. Distribution of token weights of nodes visited during PARDIS reduction.

(2) Distribution of Token Weights. The motivation behind proposing PARDIS HYBRID as discussed in Sect. 3.1 was that if lists in a test case shrink after removing nodes with large unique token weights, applying DD on list elements with the same weight can be beneficial. In fact, the more of the remaining nodes that share token weights, the more beneficial using DD becomes since it provides the opportunity to remove those nodes in just one trial. This can avoid the possibly time-consuming process of visiting nodes one by one. To understand the distribution of token weights in practice, we perform PARDIS (the one node at a time removal) on `gcc-64990` and record token weights of nodes visited during the removal process. Figure 8 shows the distribution with **5** as the median of token weights of nodes visited during the reduction. The small median motivates the use of PARDIS HYBRID in practice since it indicates that half of the nodes have one of only five different token weights and can benefit from the grouped removals.

Syntactic vs Semantic Validity: Perses and PARDIS discard *syntactically* invalid variants of the test case during reduction. However, there are also *semantically* invalid queries such as removing the declaration of a variable before removing its use. SGPR techniques cannot entirely avoid these queries since they guide the reduction process based on the syntax of the grammar. However, the priority order of PARDIS can mitigate this problem. By prioritizing by token weight, it is more likely to visit and remove uses before spending effort on declarations. One reason for this is that a higher token weight tends to mean that there are more uses beneath that node. For instance, in Fig. 3, uses of variables `a`, `b` and `c` are descendants of node ⑪ with nodes ⑧, ⑨ and ⑩ as their declarations. Pardis removes the uses by first removing ⑪ while Perses tries to remove the declarations first due to priority inversion. Hence, Pardis prunes nodes in one pass of the AST that Perses may require a fixed point mode to remove.

Threats to Validity: We evaluated Pardis on the same set of C test cases used in the evaluation of Perses. The implementation of Perses' grammar transformations and reduction is not publicly available, so we reimplemented Perses as described in its paper. Our implementation has been made available to provide a consistent platform for future work. However, the exact implementations, environmental settings and the scripts to check the property of interest can all impact the final results. For instance, the final sizes of the reduced test cases reported for the original Perses' implementation [6] are smaller than those using our reimplemented version of Perses. As discussed in Sect. 4.1, this may be because Perses' oracles allowed for undefined behavior, which can lead through smaller but invalid reduced test cases. To mitigate this problem, we made the oracles strictly prevent undefined behavior for both Pardis and Perses. Note that Pardis significantly outperforms both Perses' original implementation [6] and our reimplementation in terms of number of oracle queries.

While the techniques presented in Pardis are general in ability, our evaluation focuses on C in order to compare with Perses. Further investigation is required to claim that the performance benefits extend to other languages.

6 Related Work

The closest work to this paper is Perses [6]. Unlike Pardis, it suffers from priority inversion that adversely affects the reduction speed. Other generic test case reduction techniques are Delta Debugging (DD) [2], its $O(n)$ variant [13], and Berkeley Delta [18]. These face challenges when reducing hierarchical inputs. Several techniques focus on reducing hierarchically structured test cases [3,4,6,11,12,19,20]. Among these, only Perses is priority aware, in spite of its priority inversion. Indeed, most techniques process the input level by level. Like Pardis, Perses and Simp [20] are notable exceptions in that they can search across levels when deciding how to reduce. However, Simp is specific to SQL Queries. GTR [12] is notable in that it is trained when to apply different reduction operations. Finally, C-Reduce [5] is a tool for reducing C/C++ test cases that requires extensive domain-specific knowledge.

7 Conclusions

We have shown that the prior state of the art for test case reduction suffers from *priority inversion* and that this causes a significant increase in reduction time. We proposed priority aware reduction techniques, Pardis and Pardis Hybrid, that focus reduction effort where they can have the most impact. These techniques can speed reduction by 1.3× to 7.8× over the prior state of the art.

Acknowledgements. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Clapp, L., Bastani, O., Anand, S., Aiken, A.: Minimizing GUI event traces. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 422–434 (2016)
2. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002)
3. Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 20–28 May 2006, pp. 142–151 (2006)
4. Misherghi, G.S.: Hierarchical delta debugging. Master's thesis, University of California Davis (2007, Approved)
5. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 335–346 (2012)
6. Sun, C., Li, Y., Zhang, Q., Gu, T., Su, Z.: Perses: syntax-guided program reduction. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 361–371 (2018)
7. Kernighan, B.W., Ritchie, D.: The C Programming Language, 2nd edn. Prentice-Hall, Upper Saddle River (1988)
8. Albert, J., Giammaressi, D., Wood, D.: Extended context-free grammars and normal form algorithms. In: Champarnaud, J.-M., Ziadi, D., Maurel, D. (eds.) WIA 1998. LNCS, vol. 1660, pp. 1–12. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48057-9_1
9. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, 4–8 June 2011, pp. 283–294 (2011)
10. IBM Support, Test Case Reduction Techniques. http://www-01.ibm.com/support/docview.wss?uid=swg21084174
11. Hodován, R., Kiss, Á.: Coarse hierarchical delta debugging. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, 20–22 September 2017, pp. 194–203 (2017)
12. Herfert, S., Patra, J., Pradel, M.: Automatically reducing tree-structured test inputs. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, 30 October–03 November 2017, pp. 861–871 (2017)
13. Gharachorlu, G., Sumner, N.: Avoiding the familiar to speed up test case reduction. In: 2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, 16–20 July 2018, pp. 426–437 (2018)
14. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 336–345 (2015)
15. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013)

16. Wild, C., Seber, G.: Chance Encounters: A First Course in Data Analysis and Inference, 1st edn. Wiley, New York (1999)
17. Hodován, R., Kiss, Á.: Practical improvements to the minimizing delta debugging algorithm. In: Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, 24–26 July 2016, pp. 241–248 (2016)
18. McPeak, S., Wilkerson, D.S., Goldsmith, S.: Delta, July 2003. http://delta.stage.tigris.org/
19. Kiss, Á., Hodován, R., Gyimóthy, T.: HDDr: a recursive variant of the hierarchical delta debugging algorithm. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018, pp. 16–22 (2018)
20. Bruno, N.: Minimizing database repros using language grammars. In: Proceedings of 13th International Conference on Extending Database Technology, EDBT 2010, Lausanne, Switzerland, 22–26 March 2010, pp. 382–393, 2010