HAW
HAMBURG

Bachelor Thesis

Roxana-Teodora Mafteiu-Scai

Practical Test Case Reduction for SMT Solvers

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Roxana-Teodora Mafteiu-Scai

Practical Test Case Reduction for SMT Solvers

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science European Computer Science*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Zhen Ru Dai
Zweitgutachter: Prof. Dr. Zhendong Su

Eingereicht am: 26.06.2020

Roxana-Teodora Mafteiu-Scai

Title of Thesis

Practical Test Case Reduction for SMT Solvers

Keywords

Program reduction, Delta debugging, Debugging

Abstract

Automatic testing tools generate potentially large SMT test cases that trigger bugs. Current reducers are ineffective in reducing these test cases since they do not consider the semantics of the language while reducing it. We developed a novel framework for effective and efficient program reduction customized for failure-inducing test cases in SMT-LIB v2 format to ease testing and debugging of SMT solvers. Our framework supports domain-specific reduction rules that should not change the semantics of the program. Our evaluation results on SMT programs triggering bugs in SMT solvers demonstrate SMT-Reducer's practicality for shorter reduction time compared to the state-of-the-art C-Reduce for C/C++.

Roxana-Teodora Mafteiu-Scai

Thema der Arbeit

Practical Test Case Reduction for SMT Solvers

Stichworte

Programmreduzierung, Delta-Fehlerbeseitigung, Fehlerbeseitigung

Kurzzusammenfassung

Automatische Testwerkzeuge erzeugen potenziell große SMT-Testfälle, die Fehler auslösen. Aktuelle Reduzierer sind bei der Reduzierung dieser Testfälle unwirksam, da sie die Semantik der Sprache nicht berücksichtigen, während sie diese reduzieren. Wir entwickelten ein neuartiges Framework für eine effektive und effiziente Programmreduzierung, das speziell für fehlerinduzierende Testfälle im SMT-LIB v2-Format angepasst wurde, um das Testen und Debuggen von SMT-Solvern zu erleichtern. Unser Rahmenwerk unterstützt domänenspezifische Reduzierungsregeln, die die Semantik des Programms nicht verändern sollten. Unsere Evaluierungsergebnisse demonstrieren die Praxistauglichkeit von SMT-Reducer. Im Vergleich zum generischen C/C++ Reduzierer, der auch SMT-LIBv2 Programme reduzieren kann, reduziert SMT-Reduser SMT-LIB Programme deutlich schneller.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objective . . . 2
1.3 Outline . . . 3

2 Background **4**
2.1 SMT-LIB Format . . . 4
2.2 Program Reduction . . . 5
2.3 Reduction Frameworks and Tools . . . 6
2.3.1 Delta Debugging . . . 6
2.3.2 Hierarchical Delta Debugging . . . 7
2.3.3 C-Reduce . . . 8
2.3.4 ddSMT . . . 10
2.3.5 Perses . . . 10

3 Approach Overview **13**
3.1 SMT-Reducer . . . 13
3.2 Motivating Example . . . 14
3.3 Reduction Rules . . . 15
3.4 Iterative Approach implemented in SMT-Reducer . . . 17
3.5 Recursive Approach implemented in SMT-Reducer . . . 19

4 Implementation **22**
4.1 Parsing . . . 22
4.2 Commands List . . . 22
4.3 Reduction . . . 23
4.4 Property Checking . . . 24
4.5 Reduce Flow . . . 25
4.5.1 Iterative Approach . . . 25

4.5.2 Recursive Approach 27
4.6 Main Technologies 31
4.6.1 Python 31
4.6.2 Bash 31

5 Evaluation **32**
5.1 Benchmark Collection 32
5.2 Tools for Comparison 32
5.3 Criteria for Evaluation 33
5.4 Reduction Failure 34
5.5 Results 34

6 Conclusions **37**

List of Figures **38**

List of Tables **39**

Bibliography **40**

A Acknowledgements **44**

Selbstständigkeitserklärung **45**

1 Introduction

This chapter presents the motivation behind our work and states our primary objectives. It also describes how the thesis is overall structured.

1.1 Motivation

Satisfiability Modulo Theory (SMT) solvers, notably Z3 [18] and CVC4 [4], represent the core reasoning engines in numerous formal methods applications such as symbolic execution engines [6, 11], program synthesis [30], solver-aided programming [33], and program verification [9, 10]. However, these tools are affected by bugs like any other piece of software, and they might lead to incorrect results, which in safety-critical domains might cause significant damage.

To overcome this severe problem, the SMT community started undertaking great efforts in reporting increasingly often the SMT bugs, which caused the failure of the solvers. Often, these test cases are large and contain many elements that are insignificant for reproducing the bug. These aspects make the debugging process much more arduous and slower for the programmers and decreasing their focus. In fact, according to [15], the time needed for software maintenance is significantly higher compared to the one for other programming tasks. Hence, reducing a failure-inducing test case would be helpful before manually investigating an issue. Reducing a failure-inducing test case refers to the process of removing elements from the original test case that are unnecessary to reproduce a bug, or to simplify more complex elements to ones that are shorter or easier to understand.

The importance of reducing test cases is also stressed by the SMT solver developers themselves. In the bug tracker of the widely-used CVC4 and Z3 solvers, we found several cases in which the developers asked users to report minimal examples [23, 24, 21, 20, 22, 25]. For instance, in the issue [24], the developers of the Z3 team asked for input

reduction: "could you please reduce the files as well? It would help a lot the few Z3 folks." . Then, in [24], the developers postponed a bug fix due to its test case size and favoured the smaller ones: "without a minimalistic test case, I cannot take more seq bugs before the already reported ones or the ones with small repros have been addressed."

Nevertheless, reducing inputs in the SMT domain is mostly performed as manual work, which is considered time-consuming by the users. A number of programming techniques exist, namely Delta Debugging (DD) [36], Hierarchical Delta Debugging (HDD) [16] and C-Reduce [27]. However, applied to our domain, they do not execute as well as expected in terms of reduction efficiency, i.e. reduction time and size of reduced test case. Additionally, they may be designed for other domains than SMT.

1.2 Objective

In this thesis, we present a novel framework for effective and efficient program reduction customised for failure-inducing inputs in SMT-LIB v2 format to ease testing and debugging of SMT solvers.

Based on the framework mentioned above, a new domain-specific reducer for SMT-LIB v2 format was designed and implemented to overcome the limits of other reducers in the SMT domain. The tool supports domain-specific reduction rules that should not change the semantics of the program. Moreover, it was built on the parsing infrastructure of the state-of-the-art delta debugger ddSMT [19]. ddSMT minimises SMT benchmarks in SMT-LIB v2 format on which a given executable shows failure-inducing behaviour. It also supports all SMT-LIB v2 logics.

1.3 Outline

In Chapter 1, we present the motivation and objectives of our work.

In Chapter 2, we describe the minimal context for the thesis.

In Chapter 3, we present a high-level idea for the overall approach.

In Chapter 4, we describe the implementation, architecture and leading technologies.

In Chapter 5, we present the evaluation and the experimental results.

In Chapter 6, we describe the conclusions of this work.

2 Background

This section provides essential background information on SMT-LIB, the program reduction problem, respectively on the state-of-the-art program reduction frameworks and tools with their characteristics.

2.1 SMT-LIB Format

An SMT (short for Satisfiability-Modulo-Theories) formula is a generalisation of a Boolean instance, in which several variables are replaced by predicates from varied background theories such, e.g., quantifier free linear integer arithmetic (QF_LIA). For instance, the formula $(c \geq 1 \land e \leq c) \land d \geq e \lor a \lor (\neg b)$ contains boolean variables a, b and integer variables c, d, e. Predicates (e.g. non-Boolean variables) are evaluated, w.r.t the rules of the specific background theory. Logical operators such as conjunctions ($\land$), disjunctions ($\lor$) and negations ($\neg$), respectively quantifiers ($\exists$, $\forall$) may also be present in an SMT formula [17]. With the above, the determination of an SMT formula in certain background theories is computed by an SMT solver.

SMT-LIB (shortly for Satisfiability-Modulo-Theories Library) is proposed in [26] as the standard language for SMT solvers in order to advance their theory and practice. SMT-LIB is also used for the set of benchmarks against which to test and compare solvers. SMT-COMP [5] is an annual competition for SMT solvers, which aimed from the beginning for the adoption of the standard SMT-LIB format by SMT solver development teams. Due to this, most SMT solvers have accepted the SMT-LIB format.

The unique aspects of the SMT-LIB language related to the SMT solvers are: the underlying logic such as first-order logic, the background theory - the theory against satisfiability is verified-, the input formulas - the formula types accepted as input -, and the interface - the set of functionalities implemented by the tool.

Since its first appearance, the language has been constantly revised to meet additional needs. These may be setting various solver parameters, declaring new symbols and even asserting and retracting logical expressions, as well as exploring the satisfying assignments produced by the solver. In 2010 SMT-LIB version 2 is introduced in [7]. The authors claim that there are major backward compatibility issues between the two versions, even though SMT-LIB version 2 is based on the first one.

The SMT-LIB v2 syntax is similar to the syntax of the LISP programming language. A formula using the SMT-LIB v2 syntax is shown as an example in Listing 2.1. Working under all possible logics defined in SMT, the program first sets the information regarding SMT-LIB version and the option of enforcing model conversion. Secondly, it declares two variables a and b of type Real. Next, it adds the formula *(xor (>= b 0) (>= (+ (/ 5 b)) 2) (< a 0))* in the current assertion stack and searches for a model of all logics that satisfies this formula on the assertion stack. Similarly, it proceeds for the formula *(< (+ (∗(− 6) a)) 4)*. When both models are computed, the solver is instructed to exit the program.

Listing 2.1: A simple example formula in SMT-LIB v2 format

```
(set-info :smt-lib-version 2.6)
(set-option :solver.enforce_model_conversion true)
(declare-fun a () Real)
(declare-fun b () Real)
(assert (xor (>= b 0) (>= (+ (/ 5 b)) 2) (< a 0)))
(check-sat)
(assert (< (+ (* (- 6) a)) 4))
(check-sat)
(exit)
```

2.2 Program Reduction

An essential step in debugging and testing is program reduction. Given a program P that exhibits a particular property (e.g., a satisfiable SMT formula that is computed as unsatisfiable by an SMT solver), program reduction aims to produce a smaller program from P that maintains the same property. The reduction process continuously produces

smaller program variants and checks them against the property. The final result is represented by the smallest variant that preserves the property.

A major NP-complete problem in program reduction is computing the global minimality, according to [36] and [16]. Thus, computing the minimum result is practically limited by the program reducer's capacity. In this scope, there are two important notions: 1-minimality [36] and 1-tree-minimality [16]. 1-minimality is defined as follows: a program P is 1-minimal if any variant p derived from P by removing a single element from P no longer exhibit the property of interest. Similarly, a program P is 1-tree-minimality if the reducer cannot further simplify any node of the tree representation of P. If a reduction algorithm does not produce a 1-minimal variant, then it does not produce a 1-tree-minimal variant. This is due to the fact that the elements in the variant must also be nodes in the tree.

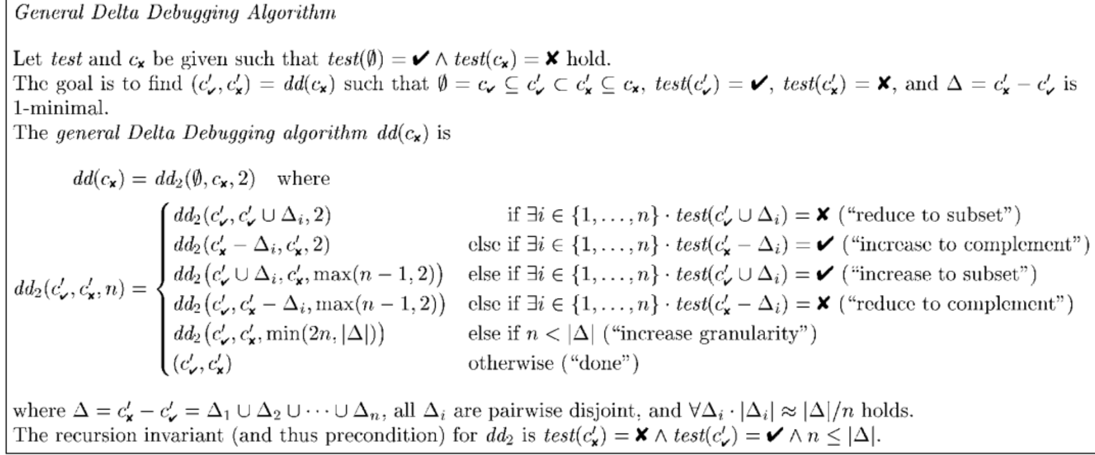
2.3 Reduction Frameworks and Tools

The state-of-the-art program reduction frameworks and tools are Delta Debugging [36], Hierarchical Delta Debugging [16], C-Reduce [27], ddSMT [19] and Perses [32].

2.3.1 Delta Debugging

Zeller and Hildebrandt proposed an algorithm called Delta Debugging [36] (shortly DD), which generalises and simplifies failing test cases to minimal test cases, respectively takes into consideration the difference between passing and failing test cases.

The algorithm is used to simplify HTML code that finally has only lines with code responsible for failure. The interest in this work is centred around the "circumstances," i.e. those whose changes may cause modified program behaviour, called changeable circumstances. In order to know the possible outcomes (success, failure, indeterminate), the attesting function was built to test the programs. The local minimums for test cases are introduced to have fewer needed tests. The binary search and a large number of smaller subsets of each test case are used to improve the chance to get a failing test case even if more subsets involve more tests. The algorithm of DD can be seen in Figure 2.1 [36].



General Delta Debugging Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $(c'_{\text{✔}}, c'_{\text{✘}}) = dd(c_{\text{✘}})$ such that $\emptyset = c_{\text{✔}} \subseteq c'_{\text{✔}} \subset c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✔}}) = \text{✔}$, $test(c'_{\text{✘}}) = \text{✘}$, and $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is 1-minimal.
The *general Delta Debugging algorithm* $dd(c_{\text{✘}})$ is

$$dd(c_{\text{✘}}) = dd_2(\emptyset, c_{\text{✘}}, 2) \quad \text{where}$$

$$dd_2(c'_{\text{✔}}, c'_{\text{✘}}, n) = \begin{cases} dd_2(c'_{\text{✔}}, c'_{\text{✔}} \cup \Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✘ (“reduce to subset”)} \\ dd_2(c'_{\text{✘}} - \Delta_i, c'_{\text{✘}}, 2) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✔ (“increase to complement”)} \\ dd_2(c'_{\text{✔}} \cup \Delta_i, c'_{\text{✘}}, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✔}} \cup \Delta_i) = \text{✔ (“increase to subset”)} \\ dd_2(c'_{\text{✔}}, c'_{\text{✘}} - \Delta_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(c'_{\text{✘}} - \Delta_i) = \text{✘ (“reduce to complement”)} \\ dd_2(c'_{\text{✔}}, c'_{\text{✘}}, \min(2n, |\Delta|)) & \text{else if } n < |\Delta| \text{ (“increase granularity”)} \\ (c'_{\text{✔}}, c'_{\text{✘}}) & \text{otherwise (“done”)} \end{cases}$$

where $\Delta = c'_{\text{✘}} - c'_{\text{✔}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |\Delta|/n$ holds.
The recursion invariant (and thus precondition) for dd_2 is $test(c'_{\text{✘}}) = \text{✘} \wedge test(c'_{\text{✔}}) = \text{✔} \wedge n \leq |\Delta|$.

Figure 2.1: The general Delta Debugging algorithm

In practice, this automated test case simplification is an integral part of other automated testings as it can be seen in next subsections [could mention other applications?].

The authors reported the results obtained with the proposed algorithm for GNU C compiler, Mozilla web browser and some UNIX utilities.

To minimise the failing test case and to have reduced time for making the difference between the failing test and the passing test, the isolation of failure-inducing differences is used instead of the simplification approach. In fact, isolation means finding the relevant parts of the test case. In the isolated algorithm, it is proposed to isolate 1-minimal differences between passing and failing test cases.

2.3.2 Hierarchical Delta Debugging

Misherghi and Su presented a technique called Hierarchical Delta Debugging [16] (shortly HDD) for minimising failure-inducing programs. In comparison to DD, HDD considers the input structure. However, this makes use of the DD in each level of a program's tree-structured input to obtain fewer, but syntactically valid tests. The domains of applicability of HDD range from programming languages, HTML/XML to video codecs and UI interactions.

For finding the parts necessary for elimination, the algorithm chooses the boundaries of the tree from bottom to top as the points for division in order to assemble the original

Algorithm 1 The Hierarchical Delta Debugging Algorithm

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

Figure 2.2: The Hierarchical Delta Debugging algorithm

structure. Before processing a tree level, the number of nodes must be computed, and each node is assigned a tag in order to set the configuration of testing. After computing the minimum configuration per tree level, the input is cleared through providing primitives, i.e. removing all irrelevant nodes from that level. The algorithm can be seen in Figure 2.2 [16]. The authors also present other algorithms derived from HDD satisfying 1-minimality or 1-tree-minimality better than HDD: HDD+ and HDD*.

The authors reported that HDD was evaluated using several bug-triggering C programs in GCC compiler, bug-triggering XML programs in the Mozilla web browser. The results of this evaluation are compared with those given by the original DD algorithm and the alternative HDD algorithms. The comparisons follow the input size (sizes or tokens), the number of property tests and the final size on the same software program for each method in part. As it can be seen in the results reported by authors, the HDD framework has better results for all indicators mentioned above or similar results compared to HDD*.

2.3.3 C-Reduce

Regehr et al. presented a novel framework called C-Reduce [27] for program reduction. For this, the generic fixpoint computation is used to invokes modular transformations for performing reductions. The authors also show two other program reducers (Seq-Reduce and Fast-Reduce) which are less efficient than C-Reduce. The tests under experimental results are obtained for programs in C language. The algorithm of C-Reduce can be seen in Figure 2.3 [27].

```
current = original_test_case
while (!fixpoint) {
    foreach t in transformations {
        state = t::new ()
        while (true) {
            variant = current
            result = t::transform (variant, state)
            if (result == stop)
                break
            /* variant has behavior of interest
              and meets validity criterion? */
            if (is_successful (variant))
                current = variant
            else
                state = t::advance (current, state)
        }
    }
}
```

Figure 2.3: The C-Reduce algorithm

By a transformation, the authors meant the iterator which traverses a test case to perform modification of the source code. The iterator consists of 3 functions: new, transform and advance. New is used to create a new transformation state object, transform is used to modify a state object if possible, and finally advance is used to iterate further to the next possible transformation. Another critical aspect of a transformation is its type, being five types of transformations. The first kind of transformation does peephole optimisations. The second one generates localised but non-contiguous changes. The third operation eliminates one or more contiguous chunks of text from the input. Then, the fourth transformation is responsible for calling external parties to improve the printing. Finally, the fifth transformation performs several C compiler like optimisations.

The authors report that C-Reduce was tested several bug-triggering C programs, and the results are compared with those given by Berkely delta, Seq-Reduce and Fast-Reduce reducers. The comparisons follow the original input size, reduction time and the final size on the same C program for each method in part. As it can be seen in the results reported by authors, the C-Reduce framework has better results for all indicators as mentioned earlier.

2.3.4 ddSMT

Aina Niemetz and Armin Biere proposed a delta debugger called ddSMT [19] for program minimisation. This delta debugger follows the authors' previous delta debugger called deltaSMT and makes use of the original DD algorithm (a divide-and-conquer strategy). The tests under experimental results are obtained for programs in the SMT-LIB v2 format. The proposed tool works in sessions, where a session consists of several simplification rules. These simplifications range from macros (command define-fun), command-level scopes (commands push and pop) to named annotations (attribute :named). In each round, the tree nodes are selected according to a specific property. Then, they are exchanged by one of the substitutions using an altered version of the original DD. This process is performed in a BSF manner. The core substitution algorithm in ddSMT can be seen in Figure 2.4 [19].

The authors reported that ddSMT was evaluated from 2 angles. The first one concerns the parsing infrastructure of the tool, for which the entire SMT-LIB v2 benchmark set was used. The second angle is the delta debugger. Its testing was performed using assembled instances and SMT-LIB v2 benchmarks as input, respectively shell scripts and two real solvers: Boolector5 and CVC46. Additionally, the authors stated that a direct overall performance of ddSMT compared to other reducers, including deltaSMT specialised for the SMT-LIB v1, was not possible at that moment.

2.3.5 Perses

Chengnian Sun et al. proposed a framework called Perses [32] for software program reduction. For this, the formal syntax is used to have steps with smaller syntactically valid variants. The tests under experimental results are obtained for programs in C and Java languages.

For the beginning, the proposed framework converts the grammar of the working programming language to the normal form (called by the authors "PNF Normalization") building a parse tree, from which the irrelevant parts are deleted. The PNF Normalisation consists of 3 steps: preprocessing, transformation and normalisation. Preprocessing is used to eliminate E productions and unreachable rules, the transformation is used to transform the code into a left recursion form, and finally, the normalisation step obtains quantified rules.

```
1  def substitute (subst_fun, superset):
2      granularity = len(superset)
3      while granularity > 0:
4          nsubsets = len(superset) / granularity
5          subsets = split(superset, nsubsets)
6          for subset in subsets:
7              nsubsts = 0
8              for item in subset:
9                  if not item.is_substituted():
10                     item.substitute_with(subst_fun(item))
11                     nsubsts += 1
12             if nsubsts == 0:
13                 continue
14
15             dump (tmpfile)
16
17             if test():
18                 dump(outfile)
19                 subsets.delete(subset)
20             else:
21                 # reset substitutions of current subset
22                 restore_previous_state()
23         superset = subsets.flatten()
24         granularity = granularity / 2
```

Figure 2.4: The core substitution algorithm in ddSMT in Python-style pseudo code

After this PNF Normalization, the main reduction follows, as it can be seen in Figure ?? [32]. That works in a parse tree w.r.t the PNF grammar form. To obtain a minimised tree that has no irrelevant nodes, the authors treat in a similar way the Kleene-Start node and Optimal Nodes. In order to reduce Kleene-Plus Node, one constraint is added to be sure that there will be at least one child in each node, to avoid syntax errors. Finally, all tree nodes are reduced with one of their compatible descendants,

The authors reported that Perses was tested using 20 large C programs, and the results are compared with those given by DD, HDD, and C-Reduce frameworks. The comparisons follow the number of property tests, reduction time and reduction speed on same software program for each method in part. As it can be seen in the results reported by authors, the Perses framework has better results for all metrics above. Regarding the Java language, the Perses results are compared with those given by HDD and are better for Perses as it can be seen in the reported results.

```
    Input: P: the program to be reduced.
    Input: ψ : ℙ → 𝔹: the property to be preserved.
    Output: A minimum program p ∈ ℙ s.t. ψ(p)
 1  best ← ParseTree(P)
 2  worklist ← {RootNode(best)}
 3  while |worklist| > 0 do
 4      largest ← GetAndRemoveLargestFrom(worklist)
 5      if largest is Kleene-Star Node then
 6          (best, pending) ← ReduceStar(best, ψ, largest)
 7      else if largest is Kleene-Plus Node then
 8          (best, pending) ← ReducePlus(best, ψ, largest)
 9      else if largest is Optional Node then
10          (best, pending) ← ReduceStar(best, ψ, largest)
11      else if largest is Regular Rule Node then
12          (best, pending) ← ReduceRegular(best, ψ, largest)
13      else continue // Skip token nodes
14      worklist ← worklist ∪ pending
15  return best
```

Figure 2.5: The algorithm for the main reduction

3 Approach Overview

This chapter presents an overview of the reducer with its different approaches, respectively several reduction rules implemented. Moreover, it shows a motivating example for our work.

3.1 SMT-Reducer

We propose *SMT-Reducer* as a program reduction framework for failure-inducing inputs in SMT-LIB v2 format. Our conceptual insight is to exploit the semantics of the SMT-LIB programs under reduction rather than their formal syntax (i.e. grammar) to generate smaller, syntactically valid variants.

The framework has a number of SMT-specific reduction rules, all of which are sound, i.e. they should not affect the semantics of the SMT formula. An example of an SMT-specific reduction rule is reducing conjunction of a term a and the implicit constant 1 (e.g. *(and a)* to the term a according to the monotone laws of Boolean algebra. A similar example applies to the disjunction of a term a and the implicit constant 1 (e.g. *(or a)*). The reduction rules alongside their type are further classified and described in 3.3.

The reduction algorithm of SMT-Reducer can work either in an iterative approach or in a recursive approach. The iterative approach sequentially applies mutations on a single state until no further mutations can be applied. In contrast, the recursive approach generates all possible combinations of reduction applications in both quantity and order. Recursively generating recursive states is more time consuming, as all possible combinations of reductions are applied in all order combinations, as well as saving each new state to disk and checking each state's solver result. At the same time, this method will use more resources for maintaining these states, both in storage and RAM. However, this approach ensures completeness as it does account for all possible outcomes, and can be extended to support fault-recovery and pause/resume functionality as all current states

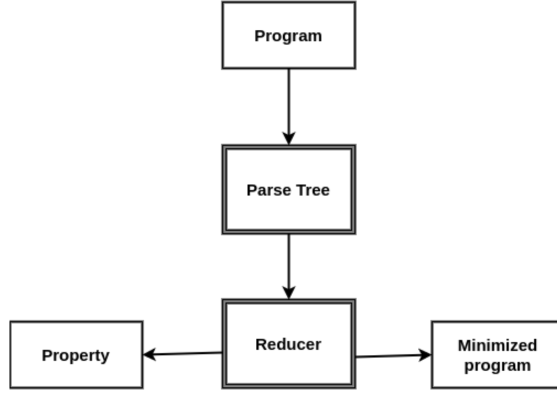


Figure 3.1: The overall workflow of SMT-Reducer

are saved. However, in contrast, the iterative variant will continuously apply reductions on the same state, thus avoiding generating all combinations of reduction and order, while still producing a valid result in the end. This procedure will result in a more efficient process in terms of time and resource usage at the expense of completeness. These approaches are further explained in the following sections.

Figure 3.1 shows the overview workflow of SMT-Reducer for reducing a SMT-LIB v2 formula considering the property of interest. Firstly, it parses the formula into a parse tree SF. Then SF is sent to the reduction flow, which contains several domain-specific reduction rules. SMT-Reducer progressively applies various rules on the tree and checks the property test. Finally, it outputs the minimized program upon termination.

It is worth note that some implementation details have been left aside in the diagrams for simplification purposes.

3.2 Motivating Example

We illustrate SMT-Reducer with a real failure-inducing SMT-LIB v2 test case reported to the CVC4 development team [28]. The full algorithm for each approach is detailed, along with the approach itself. Figure 3.2a shows a 86-byte program. Working under all possible logics defined in SMT, the program declares a variable v of type Real, and adds the formula *(and (or (= (* v v))) 0)* in the current assertion stack. Finally, it searches for a model of all logics that satisfies the current formulas on the assertion stack. As mentioned in the issue report, the behaviour of *(= (* v v) 0)* represents the root cause of

the assertion failure. Thus, we consider it the property of interest in our example. Then SMT-Reducer outputs a minimized program that eliminates the *and nesting* and prints only *(= (* v v) 0)*.

Initially, SMT-Reducer parses this formula into the parse tree, as it can be seen in Figure 3.3a. Following this, it collects from each scope all commands and their arguments, which act as tree nodes. In our example, these are *set-logic*, *declare-fun*, *assert*, respectively *check-sat*.

After this step, the algorithm begins determining for each command whether a particular reduction rule can be applied. The first node simplified is the *and* node since it is a conjunction of a term and the implicit constant 1. Finally, *or* node is reduced due to being a disjunction of a term and the implicit constant 1. In both cases, the tree containing the nesting operator is replaced with its only child maintaining the property of interest of the original input while the semantics remaining unchanged. Figure 3.2c shows the final, reduced output, whereas Figure 3.3b shows the final reduced parse tree. As shown in Figure 3.2c, the reduced program preserves the logics set, the variable declaration, and the satisfiability check. However, it has a smaller formula on the assertion stack. The size of the reduced program is 75 bytes.

3.3 Reduction Rules

Given a test case T in SMT-LIB v2 format that exhibits a property, our goal is to apply a series of reduction rules to generate the smallest variant of T that preserves the property. In our framework, these reduction rules are classified into two main categories: sound rules and unsound rules.

A sound reduction rule applied to T is highly likely to maintain the semantics of T. For instance, *(and a)* reduced to *a* represents a semantic equivalence of the two statements due to the fact that the result of both is dependent to the value of *a*. The sound reduction rules implemented in SMT-Reducer can be seen in Table 3.1.

Meanwhile, an unsound reduction rule applied to T is supposed to change the semantics of T. An example for this is deleting the command *(+ a b c)* from a formula.

```
(set-logic ALL)
(declare-fun v () Real)
(assert (and (or (= (* v v) 0))))
(check-sat)
```

(a) Original

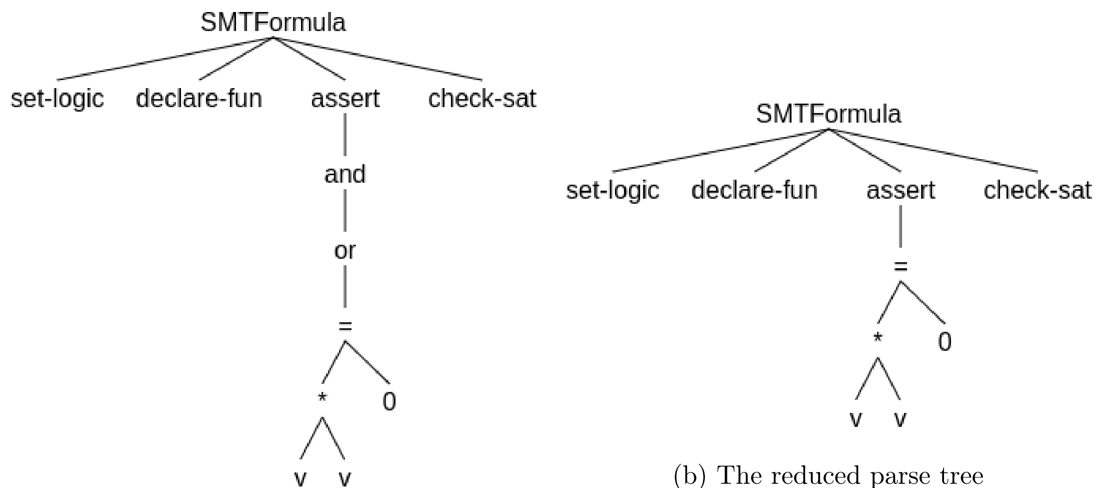
```
 (set-logic ALL)
(declare-fun v () Real)
(assert (or (= (* v v) 0)))
(check-sat)
```

(b) SMT-Reducer: first success

```
(set-logic ALL)
(declare-fun v () Real)
(assert (= (* v v) 0))
(check-sat)
```

(c) SMT-Reducer: final output

Figure 3.2: A simple example formula in SMT-LIB v2 format



(a) The original parse tree simplified for illustration purpose

(b) The reduced parse tree

Figure 3.3: The parse trees given the Example in Figure 3.2

Our tool only performs so-called “sound” reductions, that are expected to leave the program’s semantics unchanged. Our intuition is that each reduction step has a high probability of still exhibiting the property of interest. The intuition behind the and/or nesting rule, respectively the operator nesting is based on the monotone laws of Boolean algebra [2]. In the case of the negation simplifications, these are established from ordinary algebra and nonmonotone laws of Boolean algebra. According to the SMT-LIB [7], the language supports syntactic sugar for a conjunction of same-type binary functions. The binder simplifications are motivated by the compiler like transformation for eliminating unused variables.

3.4 Iterative Approach implemented in SMT-Reducer

Figure 3.4 presents the detailed steps of our approach performed iteratively. First, we parse the input in our internal representation using the ddSMT Parser [19]. This parsed input is considered the initial state, from which the set of commands is acquired. A command is treated as a tree node.

For each command, we verify the applicability of each reduction rule described in Table 3.1. If the reduction rule can be applied on the current command, then we increment the counter of applied reductions and reflect the changes on the formula. Otherwise, we attempt the same reduction on the children of the command. After applying a reduction rule, if the rule was applied at least once, we also export the variant back in SMT representation and pass it to the SMT solver, which returns the solver result. In case the solver result is different from the initial property, we restore the previous state, re-initialise the set of commands and reapply the same reduce flow to all commands.

After processing all commands, we then verify whether there was at least one reduction applied in the previous round. If this is the case, we reapply the same reduce flow to all commands. Otherwise, our formula can no longer be reduced, and we stop the reduction process. The final result is computed by returning the formula with the smallest size from all the intermediate states valid w.r.t the solver result.

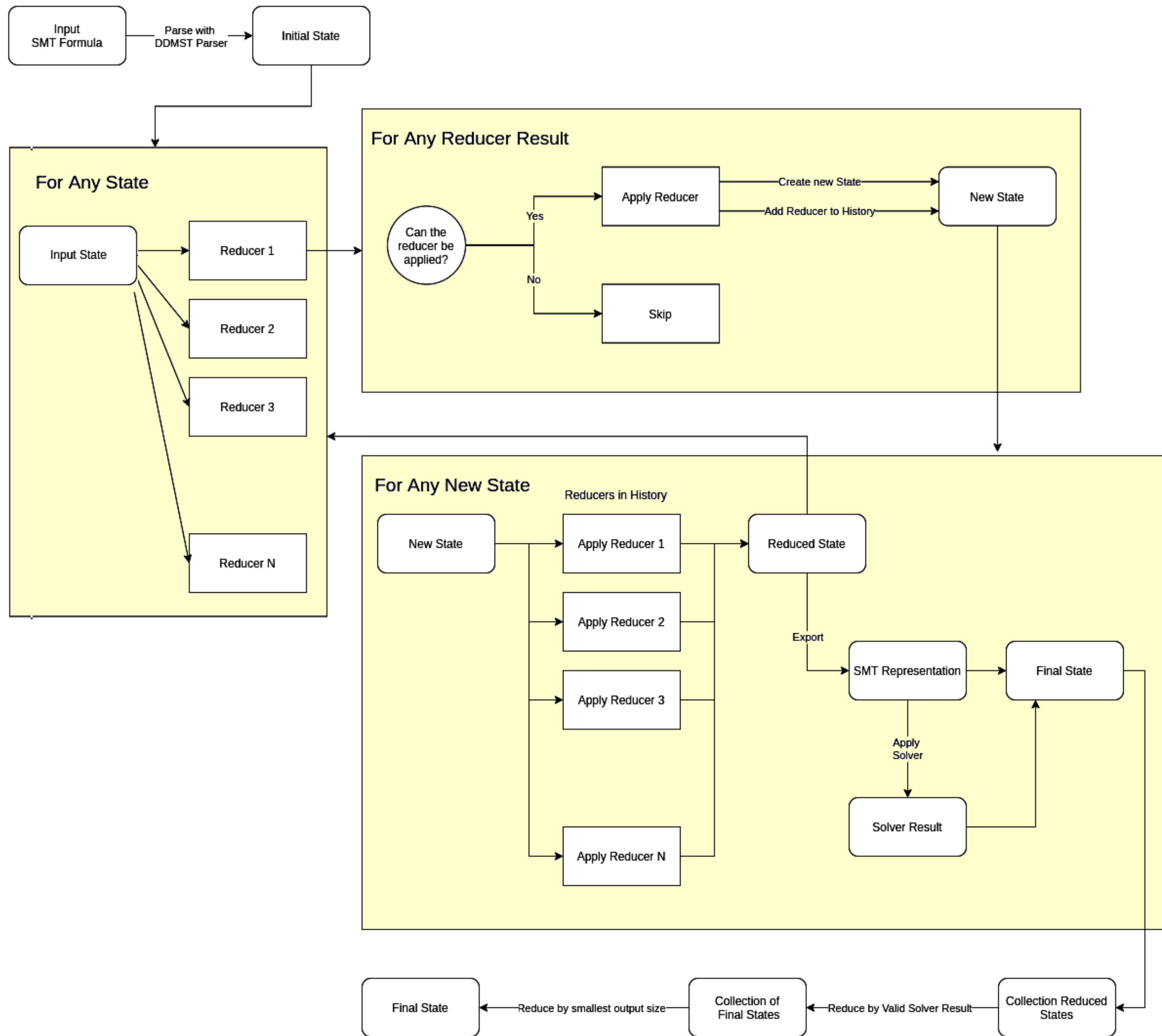


Figure 3.4: Overview of the Iterative Approach implemented in SMT-Reducer

3.5 Recursive Approach implemented in SMT-Reducer

Figure 3.5 illustrates the detailed steps of our approach performed recursively. First, we parse the input as in the iterative version. This parsed input is considered the initial state, which receives the initial input, respectively, the needed modifications. We verify for the current state the applicability of each reduction rule described in Table 3.1. If the rule can be applied, we create a new state, adding that reducer rule to its history, and the new state to the state array. After processing all reducers, we then proceed to apply the same process to any new states generated previously.

When a new state is created, the original formula is transmitted, the possible reductions are applied, and then the reduced formula is exported back in SMT representation. We pass this reduced formula to the SMT solver, which returns the solver result. In case the solver result is different from the initial property, we eliminate this state from the state array and continue with the following state.

As a final step, when there are no more possible reduction rules to apply, we reached a final state. This final state is added to the collection of final states. We compute the final result by getting the smallest size of all the final states valid w.r.t the solver result.

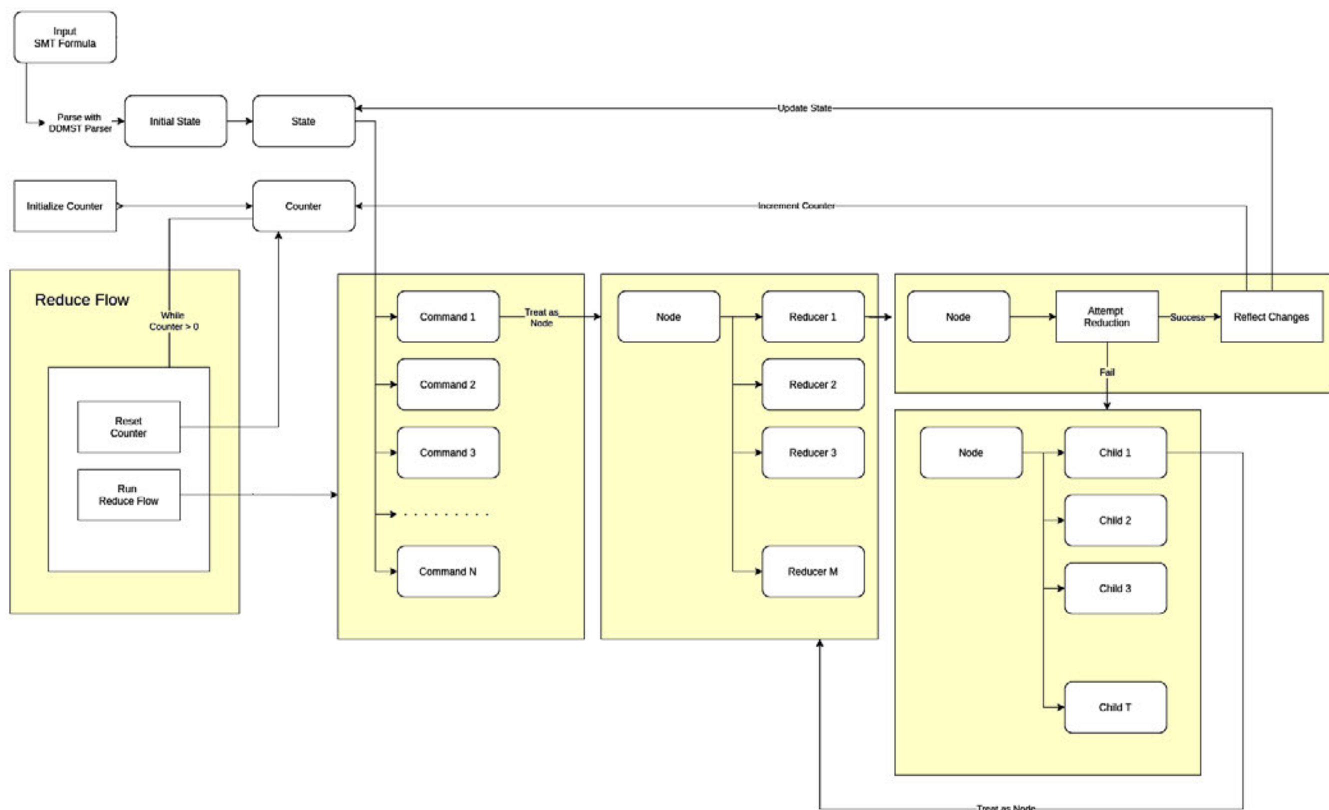


Figure 3.5: Overview of the Recursive Approach implemented in SMT-Reducer

Table 3.1: Reduction rules

Type	Reduction Rule	Example Original	Example Reduced
And/or nesting	one argument	(and a)	a
And/or nesting	more arguments	(and (and (or a) b) c)	(and a b c)
Operator nesting	(op (op <expr1> .. <exprn>) <expr3>) → (op <expr1> ... <exprn>), where op means an operator from (+, *, <, >, ≥ , ≤, =, ≠)	(+ (+ 1 2) 0)	(+ 1 2 0)
Math simplification	(* <expr1> ... <exprn> 0) → 0	(* 2 0)	0
Math simplification	(* <expr1> ... <exprn> 1) → (* <expr1> ... <exprn>)	(* 2 1)	2
Math simplification	(+ <expr1> ... <exprn> 0) → (+ <expr1> ... <exprn>)	(+ 2 0)	2
Negation	(not (not <expr>)) → <expr>	(not (not (< a b)))	(< a b)
Negation	(not <expr>), where <expr> contains a symbol of comparison from (<, >, ≥ , ≤, =, ≠)	(not (=> a b))	(< a b)
Negation	(not <constant boolean>)	(not false)	true
Syntactic sugar	(and (f a b) ... (f x z)) → (f a b ... x z), where f is a binary function	(and (+ 1 2) (+ 3 4))	(+ 1 2 3 4)
Binder simplification	elimination of binded variable(s) defined in outer scope, but not used in scope	(let ((a Int) (b Int)) (= 5 b))	(let ((b Int)) (= 5 b))

4 Implementation

This chapter presents the architecture of the main modules with their particularities, respectively, the leading technologies employed in the creation of the project.

SMT-Reducer was implemented in Python 3 and from an architectural point of view, the tool consists of five main modules described in the following sections.

4.1 Parsing

The parsing infrastructure for SMT-LIB v2 format is based on ddSMT. Additionally, we have extended the parser to recognise newer functions introduced in the latest version of SMT-LIB v2 [8]. A SMT-LIB v2 formula is internally represented as a scope tree. As in SMT-LIB v2, a scope can be either local, encapsulating variable bindings and sorted variables (defined by let, exists and forall commands), or global, as delimited by the push and pop commands. It is worth mentioning that there is no distinction between functions and variables or constants. Each scope node has the purpose of maintaining a nesting level, a subset of scope nodes, and a set of functions. Respectively, the command-level scopes contain a set of commands and a set of sorts.

The access to either currently visible sorts or functions is performed in constant time. This complexity is due to the parsing infrastructure ability to imitate the visibility of sorts and functions, similar to the compiler design techniques.

4.2 Commands List

Algorithm 1 presents the function *crawlScopes* which is responsible for acquiring the list of commands. Taking into consideration the internal representation of an input, the function receives as parameter a scope node called *scope*. Initially, the scope is the root

of tree of scopes. The function firstly generates the list of commands in the variable *cmds* of the current scope node. Then, it iterates recursively through each scope child *scp* of the scope node, gets its corresponding sub-commands and appends them to the list of commands. Finally, this list is returned.

Algorithm 1: The function for getting the list of commands

Data:
scope $\leftarrow$ *the scope node in the parse tree*
Function `crawlScope(`*scope*`)`**:**
  let $cmds \leftarrow scope.cmds$
  foreach $scp \in scope.scopes$ **do**
    $cmds \leftarrow cmds \parallel crawlScope(scp)$
  end
  return cmds

4.3 Reduction

As mentioned in Chapter 3, there are several reduction passes implemented in SMT-Reducer. The order in which we attempt these passes is: and/or nesting, operator nesting ($+$, *, $<$, $>$, $\geq$, $\leq$, $=$, $\neq$), syntactic sugar simplification, negation simplification (double negation of an expression, negation of an expression, negation of a constant boolean), math simplification (multiplication by 0, multiplication by 1, addition with 0), and binder simplification (let, forall, exists).

For the applicability of a particular type of reduction pass, we verify whether a specific condition is met either by a command node or by its parent node. If that is possible, then we apply the reduction on that command. This is performed as follows: commands are replaced with either a more straightforward expression (e.g. *(not (< a b))* $\rightarrow$ *(*$\geq$ *a b)*) or even one of their descendant (e.g. *(not (not (< a b)))* $\rightarrow$ *(< a b)*).

The pseudocode for reducing the *and/or* nesting is provided in Algorithm 2 in order to illustrate this process. Firstly, the function *detect* with the command node and its kind as parameters, detects whether an *and/or* nesting reduction can be applied to that node by satisfying three main properties simultaneously. The first property refers to the node having only one child. The second one is with regards to the type of child node that has

to be a function. The last one implies that the kind of the node has to be either AND or OR.

If the current node meets the criteria, then the function *apply* follows. We call this function alongside the current node, the index of its only child and the child itself. If the child has only one grandchild, then we substitute the child of the current node at the given index with the great-grandchild of the node. Otherwise, the child of the current node at the given index is replaced with the grandchild of the node.

Algorithm 2: The pseudocode of the and/or nesting reducer

Data:

$kind \leftarrow the\ kind\ of\ the\ tree\ node$

$node \leftarrow the\ tree\ node$

$child \leftarrow the\ child\ node$

$index \leftarrow the\ child\ index\ in\ the\ tree\ node$

Function `detect`(*node, kind*)**:**

    if *node has only one child* **and** *this child is function* **and** *kind in* $\{AND, OR\}$ **then**

        return True

    end

    return False

Function `apply`(*node, index, child*)**:**

    if *the number of grandchildren of the child node is* 1 **then**

        $node.children[index] \leftarrow child.children[0].children[0]$

    else

        $node.children[index] \leftarrow child.children[0]$

    end

4.4 Property Checking

An essential part of the program reduction problem is to check whether a partially-reduced test case still exhibits the same property of interest as the original test case provided as input. In this sense, a Bash script was created to perform the check operation of the new variant against the desired SMT solver. The pseudocode of the script can be seen in Algorithm 3. Note that this is partially configurable by the user for specifying

which solver to be used and how, and receives as arguments the path of the new variant, the path of the original solver output, respectively the path where the new result should be stored for further comparison. There are two possible return results of the checking script: 0 if the property of interest is maintained; 1, otherwise.

Algorithm 3: The pseudocode of the testing script

Data:
variant ← *the path of the variant*
original_output ← *the path of the original solver output*
final_output ← *the path where the solver output of the variant is saved*
Main `test` (*variant, original_output, final_output*) **:**
  let *solver_path* ← *the path of the solver*
  let *solver_arguments* ← *the testing script path*
  let *timeout_duration_arg* ← *the duration in seconds*
  timeout callSolver(*solver_path variant*
  if *the outputs are equal* **then**
    exit 0;
  end
  exit 1;

4.5 Reduce Flow

The reduce flow algorithm for each approach described in Chapter 3 with its main differences can be seen in the following sections.

4.5.1 Iterative Approach

In Algorithm 4, our iterative algorithm of SMT-Reducer is described. The main method *reduce* receives as parameters the parsed SMT-LIB v2 input, the path to the testing script, as well as the path to the output. Firstly, the main method prepares the environment for the reduce flow by calling the function *crawlScope* in order to obtain the list of commands *cmds*. Then, the main method starts the reducer flow by calling the *loop* function.

The *loop* function receives as parameters the list of commands *cmds*, respectively the list of functions *funcs*, each function implementing a reduction rule. We save the amount of reductions rules applied in the current attempt in the variable *cnt*. We initially assume

that there is at least one reduction rule that can be applied in order to start the first round. Therefore, the *cnt* is initialized with 1. In each round, the variable *cnt* begins at 0. Then, for each command *cmd* from our list *cmds*, we create a backup of the current state, which is stored in the variable *backup*.

Following this, we proceed with the attempts of applying each reduction rule by calling the corresponding function *fun*, which always receives as parameter the current command *cmd*. Each function *fun* computes the number of a particular reduction rule applied to the command tree. This number is stored in the variable *result*. When there is at least one application, we update the amount of reductions *cnt* for this round and call the testing script to compare the original output and the new variant output.

In case the testing script returns 1, i.e. the solver produces a different output, then we restore the state in the variable *sf* with the help our previous created backup state, as well as reinitialise the list of commands *cmds* for the new state *sf*. We also stop the current reducer attempt and restart the whole process with the new data by calling the function *loop* again. In case the testing script returns 0, we continue the attempt of applying the reduction passes on the current state. The reduce flow finishes when there

was no reduction applied in the previous round, i.e. *cnt* has the value 0, and we save the latest variant as the final output.

Algorithm 4: The reduce flow of iterative SMT-Reducer

Data:

$sf_arg \leftarrow the\ parsed\ input$

$tester_arg \leftarrow the\ testing\ script\ path$

$outfile_arg \leftarrow the\ output\ path$

Main `reduce(`$sf_arg,\ tester_arg,\ outfile_arg$`)`**:**

  let $cmds \leftarrow crawlScope(sf_arg.scopes)$ all commands from all scopes

  $loop(cmds,\ reducersList)$

Function `loop(`$cmds,\ funcs$`)`**:**

  let $cnt \leftarrow 1$ the amount of reductions applied in the current attempt

  while $cnt > 0$ **do**

    $cnt \leftarrow 0$ **foreach** $cmd \in cmds$ **do**

      let $backup \leftarrow abackup\ for\ the\ current\ state$

      foreach $func \in funcs$ **do**

        let $result \leftarrow func(cmd)$

        if $result > 0$ **then**

          $cnt \leftarrow cnt + 1$

          if *solver produces different result* **then**

            $sf \leftarrow backup$

            $cmds \leftarrow crawlScope(sf.scopes)$

            $loop(cmds,\ funcs)$

            break current reducer attempt

          end

        end

      end

    end

  end

4.5.2 Recursive Approach

The recursive approach is modelled around reduction states. Each state represents a specific chain of reductions applied to a given input. Thus, through recursion, the algorithm,

in general, produces all possible reductions in all possible orders. In Algorithm 5, the reduce algorithm of recursive SMT-Reducer is described, reduced to a single state. The main method *reduce* receives as parameters the parsed SMT-LIB v2 formula, the history list, as well as the path to the output. Firstly, the main method prepares the environment for the reduce flow by calling the function *clone* in order to create a backup formula for reduction restoration, respectively by calling the function *crawlScope* in order to obtain the list of commands *cmds*. Then, it initialises an empty list of child states which will be later updated by applying reductions. Following this, the method starts the reducer flow by calling the *apply* function for each reduction from the history list and the list of commands. Finally, the current formula is saved in an SMT representation back to the disk, alongside debug data, and the result of calling the solver with the current reduced state.

Algorithm 5: The reduce flow of recursive SMT-Reducer (state-dependant)

Data:

Given any state:

$formula \leftarrow the\ input\ formula$

$history \leftarrow the\ ordered\ list\ of\ reductions\ to\ be\ applied$

$index \leftarrow the\ identifier\ path\ in\ reduction\ graph$

Main `reduce`($formula$, $history = []$, $index = None$)**:**

- $backup \leftarrow clone(formula)$ used to restore when solver fails
- $cmds \leftarrow crawlScope(formula.scopes)$ the list of commands to be processed
- $currentIteration \leftarrow 0$ used for tracking child iterations
- $states \leftarrow []$ The list of child states created
- **foreach** $reduction \in history$ **do**
  - $reduction.apply(cmds)$
- **end**
- save current formula to disk and call solver with current state

The reduction detection part of the approach is detailed in Algorithm 6. The main function, *reduce*, will iterate through each command in *cmds*, and each reducer in *reducers*, and apply *reduceNode* with the node, reducer, as well as the current path (*index*) to the current command. After all command + node combinations have been analysed, and new states generated, the *reduce* function will then iterate through all child states and call *reduce* on them, as well, thus continuing the recursion. The *reduceNode* function, receiving a *node*, *reducer* as well as reducer arguments (*args*) and the *path* to the command, will handle detection of reduction applicability for the given node, and, if successful, will

generate a new state. It will iterate through each *child* of the current node, update the current path in *newPath* and determine the correct *detect* and *passArgs* to use in the detection process. The latter part is required as some reduction rules only rely on a single node for detection, whereas others will require both a node and its parent to correctly detect if a reduction is possible at this stage. If such a *detect* and *passArgs* combination is possible, a new *State* is created, using the *originalFormula* generated in the Algorithm 6, a history containing all current reductions, and adding a new *Reduction* with the current reducer, and an updated path for output. In the end, regardless of reduction detection, the algorithm will recurse through the command's children and attempt reductions for each child node, thus ensuring completeness.

Algorithm 6: The reduction detection in the recursive approach

Data:

Given any state:

$formula \leftarrow the\ input\ formula$

$history \leftarrow the\ ordered\ list\ of\ reductions\ to\ be\ applied$

$index \leftarrow the\ identifier\ path\ in\ reduction\ graph$

Function `reduce()`**:**

```
foreach index, node ∈ cmds do
    foreach reducer ∈ reducers do
        reduceNode((index, node), reducer, [index])
    end
end
foreach state ∈ states do
    state.reduce() recurse through states
end
```

Function `reduceNode(`*input, reducer, path*`)`**:**

```
node ← input[1]
(reducer, args) = reducer
foreach index, child ∈ node.children do
    newPath ← path || [index]
    let detect ← function to be applied for detection, either detect, or
     detectParent
    let passArgs ← parameters to be applied for detect
    if passArgs ≠ None then
        states.append(
        new State(originalFormula, history ||
         [new Reduction(newPath, reducer(...passArgs))], index ||
         currentIteration)
        )
        currentIteration ← currentIteration + 1
    end
    reduceNode((index, child), reducer, newPath)
end
```

4.6 Main Technologies

4.6.1 Python

Python [3] is a high-level, interpreted and general-purpose dynamic programming language that focuses on code readability. It was chosen as the language for our project due to the following reasons: excellent portability, multiple programming paradigms that usually involve imperative and object-oriented functional programming, respectively comprehensive and large standard library that has automatic memory management and dynamic features.

4.6.2 Bash

Bash [1], which comes from the acronym for the 'Bourne-Again SHell', is a Unix shell and command language. It was preferred due to its widely availability on various operating systems and ease of being called from the central system developed in Python in order to perform the oracle check operation.

5 Evaluation

We want to determine how well our highly-customised framework for SMT-LIB v2 format performs concerning the reduction time and size of reduced test cases compared to C-Reduce, which is a widely used reducer designed initially for C/C++, but which is also effective in reducing programs or formulas written in many other languages. We are also interested in comparing the two different approaches of SMT-Reducer against each other.

5.1 Benchmark Collection

We considered multiple test cases triggering incorrect-result bugs and solver-crash bugs. According to our observation, wrong-result bugs tend to be easier to reduce, while others (e.g. an assertion failure) are more difficult.

Therefore, we considered real large test cases in SMT-LIB v2 format generated by the SMT solver testing tools YingYang and OpFuzz [35, 34]. These triggered bugs which were reported on the official repositories of the two mainstream SMT solvers (Z3 and CVC4). We have selected 16 recent bug reports and asked the authors of the aforementioned tools for the unreduced test case. The selected bug reports cover both incorrect result and solver crash. Additionally, the selected bug reports can be reproduced w.r.t. at least one official version of the corresponding solver. We chose real-world test cases rather than artificial ones, since the tool we propose will also be used for this purpose.

5.2 Tools for Comparison

We run the iterative SMT-Reducer, as well as the recursive SMT-Reducer in comparison with C-Reduce in one-thread mode, respectively in two-thread mode. Over the state-of-the-art reducers, only C-Reduce is used in our evaluation since the SMT community

prefers using it to reduce SMT formulas due to its ability to apply a couple of general-purpose reduction rules (e.g. constant deletion).

We run the C-Reduce tool in two separate instances: once running on one thread, and once running on two threads. It is known that the default behaviour of the tool is to use as many threads as it can. However, as this may vary depending on the machine used, and cannot be consistent, and easily reproducible, we have opted to select two use cases, to determine its efficiency and viability. The first case, using one single thread, has been selected as a direct comparison with the SMT-Reducer, whereas the second case, using two threads, has been selected as the most accessible multi-threaded implementation. The goal is to observe the behaviour and efficiency differences between single-threaded and multi-threaded executions, with a direct comparison to SMT-Reducer being possible, and ensuring maximum reproducibility.

The minimum inputs to each reducer are the bug-triggering test case and the script mentioned in Chapter 4 that verifies whether a variant is successful, i.e. it maintains the property to be tested for. Additionally, SMT-Reducer requires the output path as parameter and C-Reduce requires a "thread" parameter to specify the number of thread cores on which the reduction should be performed.

All experiments were conducted on a machine with 12 GB of RAM, based on an Intel Core i7-4600U processor, running Ubuntu Linux 18.40 in 64-bit mode.

5.3 Criteria for Evaluation

We evaluated reduction efficiency in connection with reduction time and reduction size. In this section, we further quantify reduction efficiency in the following two different metrics. A reducer is considered more reduction efficient as other if the two metrics are significantly smaller.

Reduction Size This metric gives the size of the original test case and that of the minimised test by a reducer.

Reduction Time The metric aims to measure the performance time a program reducer needs to reduce a test case. This is conditional on the number of property tests executed and the time each property test takes.

Our choice of metrics —the reduction size and the reduction time— is determined by our observation that these are the main metrics that the SMT users are interested in when reporting solver bugs.

5.4 Reduction Failure

In the case of non-deterministic behaviour of a specific reducer under evaluation, we do not gain much insight into how well this performs. Therefore, the entire test case is omitted in our comparison. We observed that resource-exhaustion conditions such as timeouts and memory limits exceeded are the most common sources of non-determinism.

Other possibilities of reduction failure are that the original and reduced test cases trigger different bugs, or the reduction size of the final result remains the same as the original one.

5.5 Results

Table 5.1 shows the results of reduction efficiency in terms of time performance in seconds and size reduction in kilobytes for each deterministic test case, as well as the original size of a test case. Due to a significant variance in test case results, the mean and median values for each metric are also reported in Table 5.1.

For the first test case and last test case, it can be seen that both approaches of SMT-Reducer produced the smallest output with a 7.14% difference compared to the original. However, the iterative approach had a significantly lower performance time than the recursive approach.

If we now turn to the second test case, it can be observed that the first three reducers could not produce smaller output. Interestingly, a 0.1-kilobyte increase in the final outcomes of SMT-Reducer was recorded due to the way our tool printed the outcomes. Only C-Reduce in 2-thread mode succeeded in reducing 5.71% of the formula.

The third case, as well as the fifth case, are also compelling. All reducers led to a reduction of the original formula. The most surprising aspect of the third case lies in the small output sizes produced (46.66−60.95% shorter than the initial size) by C-Reduce. We speculate that this might be due to C-Reduce ability to perform unsound reductions such as removing one or more contiguous lines of text.

Considering the fourth case, it is interesting to point out that both executions of the C-Reduce tool have exited with no reductions applied, while SMT-Reducer succeeded in reliably reducing the input.

Another aspect worth mentioning is the time required by C-Reduce using either one thread or two threads in all results except the fourth test case. It would be reasonably assumed that running the software on two threads would produce more efficient benchmarks. However, as proven by the results obtained, this is not always the case. We assume that C-Reduce using two threads to run more complex reduction attempts in parallel. This is opposed to running the same reductions as evidenced in the one thread model, in parallel. Thus, the result is more complex, requiring higher performance time. This is also supported by the output size difference between the two threading options.

The reduced test cases by SMT-Reducer had a significant number of nested logical operators: *and*, *or*, +, and *, as well as negation operators. For instance, in the first test case, we have the following formula: *(= YhGyveT (+ (+ (* (− 196.77897150533624) shifted1v) (* (− 358.379758638266) shifted2x2)) 838.3217702720069))*. This formula is reduced to *(= YhGyveT (+ (* (− 196.77897150533624) shifted1v) (* (− 358.379758638266) shifted2x2) 838.3217702720069)))*. Another example from the same test case would be the formula *(not (= (+ (+ ?v17 (* 63 ?x2)) (* (− 57) ?x3)) 72))* reduced to *(distinct (+ (+ ?v17 (* 63 ?x2)) (* (− 57) ?x3)) 72)*. These reductions are semantically-equivalent from a mathematical point of view. Therefore, the specific rules for these operators were the most effective.

One argument for the outputs reduced by C-Reduce is given by the way the initial program was printed. In the third test case, which had the smallest outputs generated by C-Reduced, each command was printed per one line. This format might have facilitated the application of the general passes implemented in C-Reduce. For instance, the deletion of an entire line might have preserved the original bug and thus eliminated information irrelevant to the bug.

The significant time efficiency difference between all examples of SMT-Reducer and C-Reduce can be explained by the target and complexity of each tool. The latter tool has been designed for C/C++, resulting in an extensive collection of possible reduction rules to be applied. It first attempts to analyse the code as C/C++, then proceeds to attempt general reduction rules, and finally attempts domain-specific rules. Thus, the process of C-Reduce reducing SMT-LIB v2 samples would undergo a long, and complex process involving a large assortment of redundant/inapplicable rules that cannot be avoided, whereas SMT-Reducer is tailored specifically for this task. Thus, SMT-Reducer achieves a far better time efficiency.

Table 5.1: Reduction size in kilobytes and reduction time in seconds

		Iterative SMT-Reducer		Recursive SMT-Reducer		1-thread C-Reduce		2-thread C-Reduce	
Id	Initial size	Size	Time	Size	Time	Size	Time	Size	Time
1	7.0	6.5	0.52	6.5	8.0	7.0	4526.12	6.8	6215.20
2	10.4	10.5	1.47	10.5	9.43	10.4	1326.34	9.8	1100.34
3	10.5	9.7	300.11	9.7	450	5.6	10000.7	4.1	38047.09
4	10.5	9.5	0.5	9.5	1.44	10.5	0.12	10.5	0.16
5	10.5	9.7	303.40	9.7	4050.34	10.4	16372.9	10.3	23531.41
Mean	9.78	9.18	121.2	9.18	903.84	8.78	6445.23	8.3	13778.84
Median	10.5	9.7	1.47	9.7	9.43	10.4	4526.12	9.3	6215.20

6 Conclusions

In this thesis, we presented SMT-Reducer, a novel, practical program reduction framework customised for the SMT domain. We also proposed two different approaches for this framework and a series of domain-specific reduction rules that should not change the semantics of the program.

We evaluated the reducer on a select number of real-world large test cases which trigger bugs in terms of reduction time and size of the reduced program. Our empirical evaluation confirms that SMT-Reduce, using the iterative approach, can reduce a test case faster compared to C-Reduce. At the same time, C-Reduce tends to produce a smaller output size compared to each approach of our tool, with a more substantial time cost attached. If an SMT test case contains several domain-specific passes as ours, then using either approach of SMT-Reducer is recommended. However, one limitation of our implementation is that it performs only sound reduction rules.

A possible strategy of enhancing the reduction efficiency for SMT formulas would be to fuse both SMT-Reducer and C-Reduce. Precisely, we could use SMT-Reducer to apply domain-specific reductions quickly, and then run C-Reduce to apply unsound reduction rules. Similarly, Le et al. introduced in the EMI testing project [12, 13, 14, 31] a higher-order reducer, which combines C-Reduce and Berkeley Delta [29].

Our future work includes designing and implementing more sound reduction rules, as well as unsound reduction rules as described in Chapter 3, in order to decrease the reduction size further. Moreover, we intend to exploit multiple cores in order to improve the reduction time. Driven by the intuition that the reduction rules used to generate variants do not need to be embedded in the framework, we also plan to design and implement a domain-specific language such that the one has the option of writing its own reduction rules in the SMT domain for a particular test case.

List of Figures

2.1 The general Delta Debugging algorithm 7
2.2 The Hierarchical Delta Debugging algorithm 8
2.3 The C-Reduce algorithm 9
2.4 The core substitution algorithm in ddSMT in Python-style pseudo code . 11
2.5 The algorithm for the main reduction 12

3.1 The overall workflow of SMT-Reducer 14
3.2 A simple example formula in SMT-LIB v2 format 16
3.3 The parse trees given the Example in Figure 3.2 16
3.4 Overview of the Iterative Approach implemented in SMT-Reducer 18
3.5 Overview of the Recursive Approach implemented in SMT-Reducer 20

List of Tables

3.1 Reduction rules 21

5.1 Reduction size in kilobytes and reduction time in seconds 36

Bibliography

[1] *Bash*. – URL https://www.gnu.org/software/bash/. – Accessed 2020-06-07

[2] *Boolean Algebra*. – URL https://en.wikipedia.org/w/index.php?title=Boolean_algebra. – Accessed 2020-06-07

[3] *Python*. – URL https://www.python.org/. – Accessed 2020-06-07

[4] Barrett, Clark ; Conway, Christopher L. ; Deters, Morgan ; Hadarean, Liana ; Jovanović, Dejan ; King, Tim ; Reynolds, Andrew ; Tinelli, Cesare: CVC4. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings*, 2011 (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)), S. 171–177. – ISBN 9783642221095

[5] Barrett, Clark ; Moura, Leonardo de ; Stump, Aaron: SMT-COMP: Satisfiability Modulo Theories Competition. In: Etessami, Kousha (Hrsg.) ; Rajamani, Sriram K. (Hrsg.): *Computer Aided Verification*, Springer Berlin Heidelberg, 2005, S. 20–23. – ISBN 978-3-540-31686-2

[6] Cadar, Cristian ; Dunbar, Daniel ; Engler, Dawson: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USA : USENIX Association, 2008 (OSDI'08), S. 209–224

[7] Clark Barrett, Aaron S. ; Tinelli, Cesare: The SMT-LIB Standard: Version 2.0. In: *A. Gupta and D. Kroening, editors, Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010

[8] Clark Barrett, Aaron S. ; Tinelli, Cesare: TThe SMT-LIB Standard Version 2.6, URL http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf, 2017

[9] DeLine, Rob ; Leino, Rustan: BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs. March 2005 (MSR-TR-2005-70). – Forschungsbericht. – 13 S

[10] Detlefs, David ; Nelson, Greg ; Saxe, James B.: Simplify: A Theorem Prover for Program Checking. In: *J. ACM* 52 (2005), Nr. 3. – URL https://doi.org/10.1145/1066100.1066102. – ISSN 0004-5411

[11] Godefroid, Patrice ; Klarlund, Nils ; Sen, Koushik: DART: Directed Automated Random Testing. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2005 (PLDI '05), S. 213–223. – URL https://doi.org/10.1145/1065010.1065036. – ISBN 1595930566

[12] Le, Vu ; Afshari, Mehrdad ; Su, Zhendong: Compiler Validation via Equivalence modulo Inputs. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Association for Computing Machinery, 2014 (PLDI '14). – URL https://doi.org/10.1145/2594291.2594334. – ISBN 9781450327848

[13] Le, Vu ; Sun, Chengnian ; Su, Zhendong: Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Association for Computing Machinery, 2015 (OOPSLA 2015). – URL https://doi.org/10.1145/2814270.2814319. – ISBN 9781450336895

[14] Le, Vu ; Sun, Chengnian ; Su, Zhendong: Randomized stress-testing of link-time optimizers, 07 2015, S. 327–337

[15] Lientz, Bennet ; Swanson, E. ; Tompkins, Gerry: Characteristics of Application Software Maintenance. In: *Communications of the ACM* 21 (1978), 06, S. 466–471

[16] Misherghi, Ghassan ; Su, Zhendong: HDD: Hierarchical Delta Debugging. In: *Proceedings of the 28th International Conference on Software Engineering*, Association for Computing Machinery, 2006 (ICSE '06), S. 142–151. – URL https://doi.org/10.1145/1134285.1134307. – ISBN 1595933751

[17] Monniaux, David: *A Survey of Satisfiability Modulo Theory*. 2016

[18] Moura, Leonardo de ; Bjørner, Nikolaj: Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R. (Hrsg.) ; Rehof, Jakob (Hrsg.): *Tools and Algorithms for*

the Construction and Analysis of Systems, Springer Berlin Heidelberg, 2008, S. 337–340. – ISBN 978-3-540-78800-3

[19] Niemetz, Aina ; Biere, Armin: ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In: *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013), affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-9, 2013*, 2013, S. 36–45

[20] numairmansur: *Soundness bug in LIA with dom-simplify tactic #3052*. 2020. – URL https://github.com/Z3Prover/z3/issues/3052. – Accessed 2020-05-05

[21] numairmansur: *Soundness bug in QF_BV with dom-simplify tactic #3040*. 2020. – URL https://github.com/Z3Prover/z3/issues/3040. – Accessed 2020-05-05

[22] numairmansur: *Soundness bug in QF_BVFP with dom-simplify tactic #3058*. 2020. – URL https://github.com/Z3Prover/z3/issues/3058. – Accessed 2020-05-05

[23] numairmansur: *Soundness bug in QF_LIA #2871*. 2020. – URL https://github.com/Z3Prover/z3/issues/2871. – Accessed 2020-05-05

[24] numairmansur: *Soundness bug in QF_S in z3-seq #2993*. 2020. – URL https://github.com/Z3Prover/z3/issues/2993. – Accessed 2020-05-05

[25] numairmansur: *Soundness bug in UF with dom-simplify tactic #3068*. 2020. – URL https://github.com/Z3Prover/z3/issues/3068. – Accessed 2020-05-05

[26] Ranise, Silvio ; Tinelli, Cesare: The SMT-LIB Format: An Initial Proposal. In: *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning (Miami Beach, USA)*, 2003

[27] Regehr, John ; Chen, Yang ; Cuoq, Pascal ; Eide, Eric ; Ellison, Chucky ; Yang, Xuejun: Test-Case Reduction for C Compiler Bugs. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, Association for Computing Machinery, 2012, S. 335–346. – URL https://doi.org/10.1145/2254064.2254104. – ISBN 9781450312059

[28] ROXAAAMS: *Assertion failure using –sygus-inference 2 #3200*. 2019. – URL https://github.com/CVC4/CVC4/issues/3200. – Accessed 2020-05-05

[29] SCOTT MCPEAK, Daniel S. W. ; GOLDSMITH, Simon: *Berkeley Delta*. – URL https://github.com/Z3Prover/z3/issues/3040. – Accessed 2020-06-07

[30] SOLAR-LEZAMA, Armando: *Program Synthesis by Sketching*. USA, Dissertation, 2008

[31] SUN, Chengnian ; LE, Vu ; SU, Zhendong: Finding Compiler Bugs via Live Code Mutation. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Association for Computing Machinery, 2016 (OOPSLA 2016). – URL https://doi.org/10.1145/2983990.2984038. – ISBN 9781450344449

[32] SUN, Chengnian ; LI, Yuanbo ; ZHANG, Qirun ; GU, Tianxiao ; SU, Zhendong: Perses: Syntax-Guided Program Reduction. In: *Proceedings of the 40th International Conference on Software Engineering*, Association for Computing Machinery, 2018, S. 361–371. – URL https://doi.org/10.1145/3180155.3180236. – ISBN 9781450356381

[33] TORLAK, Emina ; BODIK, Rastislav: A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2014 (PLDI '14). – URL https://doi.org/10.1145/2594291.2594340. – ISBN 9781450327848

[34] WINTERER, Dominik ; ZHANG, Chengyu ; SU, Zhendong: *On the Unusual Effectiveness of Type-aware Mutations for Testing SMT Solvers*. 2020

[35] WINTERER, Dominik ; ZHANG, Chengyu ; SU, Zhendong: Validating SMT Solvers via Semantic Fusion. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2020 (PLDI 2020), S. 718–730. – URL https://doi.org/10.1145/3385412.3385985. – ISBN 9781450376136

[36] ZELLER, Andreas ; HILDEBRANDT, Ralf: Simplifying and Isolating Failure-Inducing Input. In: *IEEE Trans. Softw. Eng.* 28 (2002), Februar, Nr. 2, S. 183–200. – URL https://doi.org/10.1109/32.988498. – ISSN 0098-5589

A Acknowledgements

I would like to express my gratitude to Prof.Dr. Zhendong Su for offering me the opportunity to complete my bachelor's thesis research in the Advanced Software Technologies (AST) Lab at ETH Zurich, under his supervision, as well as the unwavering guidance of postdoctoral researcher Dr. Manuel Rigger. I also very much appreciate the assistance of the PhD students: Dominik Winterer and Chengyu Zhang for the evaluation of the tool.

I would further like to highlight my gratitude to my supervisor Prof.Dr.-Ing Zhen Ru Dai for her unrelenting support and encouragement in the Computer Science Department at HAW Hamburg during my exchange mobility, which was part of the European Computer Science double degree programme. It is highly appreciated.

I would like to extend my since thankfulness to Prof.Dr. Daniela Zaharie for her unparalleled support throughout the completion of my undergraduate studies in the Computer Science Department at West University of Timisoara.

Last, but not least, I would like to thank my family, educators and friends for always inspiring and encouraging me to push my limits and widen my ambitions. Moreover, I am grateful to my parents for awakening my passion for Computer Science.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: MATTEIU-SCAI

Vorname: ROXANA - TEODORA

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Practical Test Case Reduction for SMT Solvers

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

22.06.2020

Ort — Datum — Unterschrift im Original