Normalization of SMT-LIB scripts

Kristian Ionescu
Abstract

Normalization of SMT-LIB scripts

Kristian Ionescu

Satisfiability modulo theories (SMT) is about determining the satisfiability of logical formulas over a range of one or more theories. SMT solvers are programs that are designed to determine the satisfiability of logical formulas and to find a satisfying model. Since the SMT problem is known to be NP-complete, the SMT initiative holds annual competitions to further advance and stimulate the techniques and tools used by the SMT community. To prevent unfair entries to the competition, a benchmark scrambler with a random seed is used to rename variables, randomly permutes arguments and hides benchmark names for each of the formulas. This paper is about researching the possibility of proving satisfiability of expressions that are semantically equal but not necessarily syntactical. The purpose of this project was to develop a computer program that is able to normalize benchmark scripts to a standard defined format such that by scrambling syntactically equal scripts would result in the same result when normalized.

The implementation that was conceived is incomplete and inefficient for larger SMT scripts in terms of computational complexity due to time constraints however, the research of that was conceived during this thesis provides an interesting insight in the complexity of normalizing benchmark scripts and its relation to graph isomorphism.
Contents

1 Introduction 1
  1.1 Prelude .......................................................... 1
  1.2 Background description ............................................. 1
    1.2.1 Example ....................................................... 3
  1.3 Case and purpose .................................................. 4
  1.4 Goals ................................................................ 5
  1.5 Literature study .................................................... 5
  1.6 Boundaries ............................................................ 6
  1.7 Method ............................................................... 7
    1.7.1 Evaluation ....................................................... 7
    1.7.2 Study of legacy code ............................................ 8
    1.7.3 Flow of the benchmark scrambler ......................... 8

2 The SMT-LIB Language 9
  2.1 Syntax and Semantics of SMT-LIB ................................. 9
    2.1.1 Tokens ......................................................... 9
    2.1.2 S-expressions .................................................. 10
    2.1.3 Expressions .................................................... 11
    2.1.4 Identifiers ..................................................... 11
    2.1.5 Commands ...................................................... 12
    2.1.6 assert command ................................................ 12
    2.1.7 check-sat command .......................................... 13
    2.1.8 Other commands ............................................... 13
  2.2 Benchmark scrambler ................................................ 13
    2.2.1 Description .................................................... 14
    2.2.2 Example ....................................................... 16

3 Normalization of SMT scripts 17
  3.1 Prelude ............................................................. 17
  3.2 Sorting assertion statements and arguments ..................... 17
  3.3 Variable renaming .................................................. 19
    3.3.1 Prelude ........................................................ 19
    3.3.2 Approaches ..................................................... 19
    3.3.3 Final implementation ....................................... 21
  3.4 Sorting of arguments ............................................... 25
4 Implementation ............................................. 28
  4.1 Prelude ................................................. 28
  4.2 Data representation .................................... 28
  4.3 Implementation ......................................... 29
    4.3.1 Sorting expressions ................................ 29
    4.3.2 Variable renaming .................................. 32
    4.3.3 Normalizer algorithm ............................... 33

5 Evaluation ............................................... 36
  5.1 Software requirements .................................. 36
  5.2 Program compilation ................................... 36
  5.3 Program execution ..................................... 36
  5.4 Testing ................................................. 37
    5.4.1 Code refactoring legacy code ....................... 37
    5.4.2 SMT Benchmarks .............................. 38

6 Conclusions .............................................. 39
  6.1 Summary ................................................. 39
  6.2 Future work ............................................ 41

References ................................................. 42
Chapter 1

Introduction

1. This chapter serves as an introduction to the project. Aims, objectives and issues are reported in this chapter. The background description gives an insight on the purpose of the project.

1.1 Prelude

The purpose of the thesis was to develop a computer program that can normalize mathematical and logical expressions to a defined format. The prerequisite requirements for this program is that it conforms to the standard language defined by the international initiative called SMT-LIB [2].

1.2 Background description

Satisfiability modulo theories (SMT) [4] is a category in automated deduction that studies methods for examining the satisfiability of first-order expressions with respect to some logical theory (the theory of integers, floating point or even data structures). SMT is an extension of the boolean satisfiability problem (SAT) of determining if there exists at least one interpretation that satisfies a logical expression.

SMT solvers are programs that can be used to prove validity or satisfiability of first-order expressions for a large set of logical theories. The problem of deciding the satisfiability of first-order expressions is NP-complete as it is hard to find efficient solutions for solving quantifier free formulas as well as formulas that contain bound quantifiers. Since the problem is NP-complete, the focus over recent years has been on developing efficient SMT solvers. The SMT initiative [2] holds annual competitions [1] where SMT solvers try to solve a large set of benchmarks. The competition was brought to its inception with the hope that it would further stimulate the techniques and tools used by the SMT
community to further improvements in solver implementations.

To prevent any form of cheating, a benchmark scrambler [3] is used to hide benchmark names, rename and randomly permute arguments and commutative operators of expressions given a random seed. If a SMT solver would however, solve benchmarks by identifying the semantic equivalence of pre-solved benchmarks, the program could then “solve” benchmarks by returning results of already known benchmarks. The means of identifying a semantic relation between benchmarks can be achieved by normalizing the benchmarks to a defined standard form.

A program that can normalize SMT benchmark could however have other interesting applications in the area of automated deduction, mathematical logic and computer science. One of the application domains for SMT is in the verification, analysis and construction of computer programs. The benchmark normalizer could be used as a means to identify similar constraint problems.
1.2.1 Example

Assume that we have the following scripts written in the SMT-LIB syntax.

script 1:

(set-logic QF_LIA)

(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun c () Bool)
(declare-fun d () Bool)

(assert (let ((x (not a)) (y (not b))) (and x y)))
(assert (xor a b))
(assert (not (and c d)))
(assert (not (or b d)))

(check-sat)
(exit)

script 2:

(set-logic QF_LIA)

(declare-fun n9 () Bool)
(declare-fun n8 () Bool)
(declare-fun n7 () Bool)
(declare-fun n6 () Bool)

(assert (not (and n7 n6)))
(assert (xor n9 n8))
(assert (let ((a (not n9)) (b (not n8))) (and a b)))
(assert (not (or n8 n6)))

(check-sat)
(exit)

These scripts are semantically equivalent since they only differ in variable names and structure. The SMT normalizer computes a normal form for the script such that semantically equivalent scripts will generate the same output when normalized. The result of normalizing script 1 and script 2 respectively is the following.
Script 3:

(set-logic QF_LIA)

(declare-fun v0 () Bool)
(declare-fun v1 () Bool)
(declare-fun v2 () Bool)
(declare-fun v3 () Bool)

(assert (let ( (v4 (not v0)) (v5 (not v1))) (and v4 v5)))
(assert (xor v0 v1))
(assert (not (and v2 v3)))
(assert (not (or v1 v3)))

(check-sat)
(exit)

1.3 Case and purpose

Purpose

The purpose of this project was to develop a computer program that is able to normalize benchmark scripts to a standard defined format. The resulting transformation should conform to the standard language [2] defined by the SMT-LIB initiative.

Case

The program will be used to normalize benchmarks that are typically used in the SMT competition for SMT solvers. Another use of this program is to reduce redundancy in caches by identifying cache lines with similar content through normalization, this could possibly result in fewer cache misses. It could also be used as a replacement policy for caches, the problem of predicting which cache entry is least likely to be used in the near future, instead the normalizer could identify cache lines with similar context and evict redundant cache lines.
1.4 Goals

The objectives that this report will answer:

- How are SMT expressions normalized?
- How are SMT definitions renamed?
- How are expressions sorted in the resulting transformation?

1.5 Literature study

During the beginning of the project, literature study was conducted in order to understand the syntax and semantics of the SMT-LIB language [2]. Further studies was conducted on the public benchmark library [2] to understand how the language is used in practice.

Research on Satisfiable modulo theory and its applications [4] was conducted to gain further understanding on the topic and the importance of the real life applications. The article discusses how SMT solvers are important for program analysis, testing and verification because of its mathematical properties. Important examples are discussed to explain how SMT can be applied in practice such as the job-shop scheduling optimization problem.

During the late stages of the project, research related to the field of computational complexity theory [5] was necessary. The problem of designing the normalizer algorithm exhibited properties typical of problems belonging to the NP complexity class. The problem was to prove this claim, The book "The Graph Isomorphism Problem and Approximate Categories" [5] helped to understand that the problem of normalizing similar SMT expressions is closely related to the problem of determining whether two finite graphs are isomorphic.

The source code for the SMT scrambler [3] was also studied carefully since it is providing a foundation for the normalizer to be built upon. Knowledge of the C++ language was known before the start of the project however, the book "Effective C++" [14] proved to be useful because the author explains how to write software that is more efficient, robust and more reusable. The book is really good since there is a lot to learn about C++ designs and language traps.

The Book "Beginning Linux Programming" [13] is a comprehensive book that is purposed for developers who want to learn more about C/C++ programming in a linux environment furthermore, it explains shell scripting in unix and linux environments and how it could be used to compile and build programs faster.
"First-Order Logic and Automated Theorem Proving" [7] is a comprehensive book on formal logic. The book is aimed for computer scientists that are interested in the area of automated theorem proving and gives an introduction to the syntax and semantics of propositional logic, the book provides examples and proof for various normal forms that are relevant to the thesis such as conjunctive normal form and disjunctive normal form. A propositional formula is in conjunctive normal form (CNF) if it is a conjunction of clauses in which the clauses are a disjunction of literals.

Likewise, a propositional formula is in disjunctive normal form (DNF) if it is a disjunction of clauses in which the clauses are a conjunction of literals. In the end, these techniques for normalizing propositional formulas have not been implemented in the SMT normalizer because the normalizer uses a different method of normalizing each asserted statement in the script by sorting clauses based on operator and renaming of variables which are described in later chapters.

Traditional normal forms such as DNF or CNF would not be ideal for recognizing equivalence in SMT scripts since the SMT language deals with various logical and arithmetic operators as well as different theories. It would be a difficult task to recognize which normalization procedure to apply based on these parameters so a general approach for normalizing SMT statements was implemented instead.

During the implementation stages of the project, the normalizer had to sort statements and assertions according to a defined rule. The book "Ordered Sets" [8] provides definitions of ordered sets in the field of set theory but more importantly, it describes the definition of lexicographical ordered sets which was used to sort assertions and variable statements in SMT scripts.

The book "Xcode Tools Sensei" [17] mainly covers programming of Mac applications however, the book presented useful tools for testing code coverage.

The book "Object-Oriented Programming Using C++" [6] provides a solid foundation in C++ programming. The book uses clear examples to teach both the syntax of the C++ language and sound programming principles. The book gives an overview of object-oriented programming and C++, and then builds upon this knowledge to teach complex concepts such as inheritance, templates, handling exceptions, and forward declarations of functions which was needed during the implementation stage of this project.

Research on related topics or similar work was also done during this part of the project.

1.6 Boundaries

The goal of this project is to research and develop a functioning software that can normalize scripts written in the SMT language with the hopes that it will provide a basis for further research on the topic, extensions and/or optimizations of the resulting software.
1.7 Method

This chapter describes the methodology used before the implementation of the program. This section will describe the research and decisions that were made before implementation.

1.7.1 Evaluation

During the early stages of the project, research on finding appropriate front-ends was conducted in order to find a suitable platform for the project. There exist a lot front-ends for SMT-LIB in a variety of different languages. It was decided that the project would use the benchmark scrambler as its building platform for reasons that would prove to be beneficial for the implementation of the project.

It was also important to research the SMT-LIB language. The language contains parts which are not relevant to the project such as functionality that communicates with a given SMT solver. The important parts about the language was syntax formation and the semantics behind formed expressions.

The scrambler already contains a functional parser that can be used in the project, this allowed for more focus on the main issues that needs to be resolved during implementation. The design of the source code was very modular which made it easy to recycle and modify source code that could be included in the implementation. Documentation on the benchmark scrambler was not complete however, the comments contained in the source code were sufficient to understand the purpose of ambiguous code blocks.

The scrambler is built on C++ which is a suitable programming language for this project as it contains a rich library of functions and is also known to be a high performance language. The language is also very popular, so there exist a lot of additional tools that can help with benchmarking and debugging of source code.
1.7.2 Study of legacy code

Before the source code for the benchmark scrambler can be used as a building platform for the project, it is important to understand how it was built. The source code of the benchmark scrambler is fairly large and is divided into several files. A easy way to understand the flow of the benchmark scrambler is to debug the code by inserting breakpoints at key points in the source code and use the stack trace to understand the flow of the function calls.

1.7.3 Flow of the benchmark scrambler

The benchmark scrambler generates a parser by using the GNU Bison tool [9] and ”parser.y” as its input. The file ”lexer.l” contains the context-free grammar for the SMT-LIB language and is used together with the parser generator to create a parser ”parser.cpp” and the header file ”parser.h”. During program execution, the parser reads a input script and builds a abstract syntax tree for each defined command in the script using functions defined in ”scrambler.h”. All the AST are saved in a vector of commands with the appropriate index value.
Chapter 2

The SMT-LIB Language

This chapter describes the important data structures and functions of the benchmark scrambler. Since the benchmark scrambler is provided as a building platform for this project, it is important to understand how the benchmark scrambler transforms benchmark scripts.

2.1 Syntax and Semantics of SMT-LIB

The SMT-LIB initiative is an effort, supported by various research groups, with the goal of producing an exhaustive library of benchmarks and to promote the adoption of a standard language and interface for SMT solvers and tools. SMT-LIB adopts a single and general first-order (sorted) language in which to write logical formulas. SMT scripts contain a collection of expressions and definitions and are saved with the .smt2 extension.

2.1.1 Tokens

Tokens are the basic building blocks in expressions. There are different kinds of tokens and not every sequence of SMT-LIB characters is a valid token. Although some tokens might have an intuitive meaning such as <numerals>, their semantics is dictated by the underlying logic that is being used. As an example, <binary> literals might be a bit-sequence or integers depending on the defined logic.
Here are examples of different kinds of tokens:

- **<numerals>** Numeric literal expressed as a sequence of digits (no leading zeroes).
- **<decimals>** Numeric literal followed by a decimal point followed by more digits.
- **<binary>** Prefixed by #b, expresses a single sequence of 1’s and 0’s.
- **<hex>** Prefixed by #x, denotes a sequence of hexadecimal digits.
- **<string>** Printable ASCII character string in double quotes.
- **<symbol>** A non-empty sequence of letters, digits and the characters + - / * = % ? ! . $ ∼ & ∧ < > @ that does not start with a digit.
- **<keyword>** The character ‘:’ followed by a non-empty sequence of letters, digits and the characters + - / * = % ? ! . $ ∼ & ∧ < > @.

It is important to note that some character sequences are reserved words for the SMT-LIB language. This means that they may only be used in specific contexts and are not allowed to be used as symbols or identifiers. Symbols are tokens which are used for attribute values, sort, variable, logic, theory, constant and function names.

A symbol can be any sequence of symbol characters that does not start with a digit and is not a reserved word. Some symbols such as (+ - *) have an intuitive meaning as function symbols but ultimately the semantic is again defined by the underlying logic that is being used.

### 2.1.2 S-expressions

The input and output from a SMT solver is a series of s-expressions written using SMT-LIB characters (except comments). An S-expression is either a token or a sequence of s-expressions encapsulated in a pair of left and right parenthesis for each s-expression. Tokens within a s-expression are separated by whitespace.

The grammar for building an s-expressions using tokes is defined below:

```
<spec_constant> ::= <numeral>| <decimal>| <hexadecimal>| <binary>| <string>
<s_expression> ::= <spec_constant>| <symbol>| <keyword>|  
                   ( <s_expression>* )
```

Here are some examples of various s-expressions:

- `abc ; A single token`
- `( ) ; Empty S-expression`
- `(+ a b) ; S-expression with three tokens`
- `(+ 3 (* 4 5)) ; Nested S-expression`
- `(not A) (not B) ; Two successive S-expressions`
2.1.3 Expressions

SMT-LIB expressions (<expr>.) are S-expressions built recursively from a set of basic forms commonly consisting of symbols, literals and identifiers. As an example, the expression (+ 3 5 a) consists of two literals, one identifier (a) and the operator symbol (+). Examples and uses of complex expressions can be found in the SMT-LIB documentation.

The following is an example of a SMT script:

```
(set-logic QF_UF)
(declare-fun c0 () Bool)
(declare-fun E0 () Bool)
(declare-fun f0 () Bool)
(declare-fun f1 () Bool)

(assert (xor f0 f1))
(assert (and f0 f1 c0))

(check-sat)
(assert (and c0 E0 (not f1)))

(check-sat)
(exit)
```

First and foremost, theory declaration and function symbols are defined at the beginning of the script. The set-logic command is a theory declaration used to initialize the SMT solver with the specified logic. A theory defines the relevant types of sort and functions. As an example, the theory of integers contains the sorts Int and Bool while also defining that literals are of sort Int and should be interpreted as integers. All of the standard logics are defined in the SMT-LIB documentation.

2.1.4 Identifiers

SMT-LIB requires all identifiers to be declared before being used. The declare-fun command is used to declare new symbols. To declare a new function symbol, we need to specify the sorts of the arguments and of the return value. The sort that is associated to well-formed expressions can be seen as a type in other contexts.

The command has the following form:

```
(declare-fun <symbol> ( <sort-expr>* ) <sort-expr>)
```

In the example above, the identifiers declared in the beginning of the script has no arguments and returns a bool sort. For simplicity reasons, we will refer to identifiers as variables in the forthcoming chapters.
2.1.5 Commands

Commands are reserved keywords that are used to instruct an SMT-LIB compliant solver, they can also be used to gain information about the state of the solver. As described in the previous section, the set-logic command instructs the solver to evaluate asserted expressions according to a theory. The fundamental commands that are often used in SMT compliant scripts are assert and check-sat. The output generated by a command is an s-expression, most commands will produce either the token ‘success’ or an error message with a stated reason.

The error messages are defined as s-expressions with the token ‘error’ followed by a string sequence explaining the reason. Some commands might respond with the token ‘unsupported’. This indicates that the solver does not support the command since it is defined as an optional command in the SMT-LIB language. SMT solvers are not required to support optional commands.

2.1.6 assert command

The assert command is used to instruct a solver to assume that a well-formed logical expressions is true.

The command has the following form:

(assert <expr>)

in which <expr> is a well-formed expression of sort ‘Bool’. It is important to note that the set-logic command needs to be declared beforehand, since the logic defines the necessary sort and function symbols used in the expression.

Assert statements return error in the following cases:

- The statement is malformed.
- The expression is not of the sort ‘Bool’.
- The statement occurs before a set-logic statement or if the statement is not allowed in the current logic.

Otherwise, the statement returns success, this expresses that the statement is a part of the solver context.
2.1.7 check-sat command

The **check-sat** command is used to check satisfiability of a set of assert statements. Once a set of assert statements has been declared, the **check-sat** command instructs the SMT solver to test the set for satisfiability.

The command has three return values:

- **sat**: The set of assertions are satisfiable, there exists a set of assignments of values to constants and identifiers under which all expressions evaluate to true.

- **unsat**: The set of assertions are not satisfiable, there exists no set of assignments of values to constants and identifiers under which all expressions evaluate to true.

- **unknown**: The solver cannot determine for a fact whether the set of assertions are satisfiable or not. The reason for this might vary. The solver might have just run out of time or memory.

The solver might have found a assignment which appears to satisfy all the assertions, but it cannot determine this for a fact, because the assertions might contain quantified expressions. This case might occur since the solver might not be sure if it has instantiated the quantified expression for all cases relevant to the satisfiability problem.

2.1.8 Other commands

SMT-LIB offers a set of optional commands that can be used in various cases. Depending on the return value of the **check-sat** command, additional information can be gained about the satisfiability problem. The commands **get-assignment** and **get-value** can be used to get more information about the assignment which satisfies the assertion set. In contrast, the **get-proof** and **get-unsat-core** command can be used to find out more about the unsatisfied assertion set.

2.2 Benchmark scrambler

The benchmark scrambler renames variables, randomly permutes arguments and hides benchmark names for each expression defined in the SMT script. The output is a new benchmark script that can appear syntactically different but is semantically equivalent to the input script.
2.2.1 Description

The scrambler renames variable names by using a hashmap together with the parser. When variables are parsed, they are mapped to a set of output variables. For example, if the script would contain the following set of variables \( \{v_1, \ldots, v_n\} \), each variable is mapped to the output variables \( \{x_1, \ldots, x_n\} \).

```c
void set_new_name(const char *n);
```

`set_new_name` renames input variables by using the `make_name` function. The input variable and the renamed variable are then stored in a hashmap. This forms a one-to-one relationship where the input value is used as a reference to return the renamed reference as a value.

```c
std::string make_name(int n);
```

The benchmark scrambler generates new variable names by appending an incremental integer to the variable name. `make_name` takes the current integer value, increments this value by one and appends the variable name to the integer. It then returns this as a result value.

```c
std::string get_name(const char *n);
```

`get_name` takes the input variable as its parameter and performs an iterative search on the hashmap. When a match is found, the corresponding return value is returned otherwise, the function returns the input variable.

Arguments within command declarations are stored in a vector of nodes. The structure of a node is described by the following structure.

```c
struct node {
    std::string symbol;
    std::vector<node *> children;
    bool needs_parens;
    ...
}
```

Each element in the vector is a node, the symbol can represent a whitespace character, operator or a variable. When a symbol only contains a whitespace character, the node represents an expression encapsulated in parentheses. The expression can then be found in the children vector. The parser might also remove unnecessary parentheses by the use of the `needs_parens` flag.

```c
bool is_commutative(node *n);
```

The arguments within the list are then shuffled if the operator is commutative. this check is done using the `is_commutative` function. The function also makes sure that if a certain logic has been defined in the script since a commutative operator might not be so in a different logic.

```c
bool flip_antisymmm(node *n, node **out_n);
```
Since some expressions are anti-symmetric, the function uses the opposite equivalent operator to represent the same semantic equivalent equation. This function is needed since arguments are being swapped in the expression. As an example, if a script contained the following expression

\[ a + b + 2 > 3 \]

The scrambler will recognize that the "greater than" operator is anti-symmetrical and will use the opposite "less than" operator after the left-hand-side expression is swapped position with the right-hand-side of the equation. It will produce the following result:

\[ 3 < a + b + 2 \]

However, the function is not applied on scripts that are defined as difference logic which is used for boolean combinations of in-equations of the form \( x - y < b \) where \( x \) and \( y \) are integer variables and \( b \) is an integer constant.

As mentioned in the previous chapters, the benchmark scrambler constructs an abstract syntax tree for each command and assertion that is parsed as illustrated in figure 2.1. Presented below is an example of an abstract syntax tree that the benchmark scrambler has constructed for the following assertion statement. Each rectangle represents a node, the black line that connects the upper nodes with the lower nodes denote the parent/child relationship between the nodes. Since a node has a vector of children, a node could possibly have multiple outgoing arrows.

**Expression:** \( 3x + 5y < 4 \)

\[ (\text{assert (} > ( + ( * 3 x ) ( * 5 y )) 4 )) \]

![Figure 2.1: The abstract syntax tree constructed by the benchmark scrambler.](image)
2.2.2 Example

The following is an example of a script with no logic defined. This will allow the benchmark scrambler to randomly scramble arguments of operators that are intuitively commutative. The example below shows the scripts that the benchmark scrambler generates from the same input script after it has been called three times with a random seed.

Input script:

```
(declare-fun p () Bool)
(declare-fun q () Bool)
(declare-fun r () Bool)
(declare-fun s () Bool)

(assert (let ((.def_10 (not p))) .def_10))
(assert (xor r s))
(assert (not (and r s)))
(assert (and p q r))

(check-sat)
(exit)
```

Output scripts:

| (declare-fun x1 () Bool) | (declare-fun x1 () Bool) | (declare-fun x2 () Bool) |
| (declare-fun x2 () Bool) | (declare-fun x2 () Bool) | (declare-fun x4 () Bool) |
| (declare-fun x3 () Bool) | (declare-fun x4 () Bool) | (declare-fun x1 () Bool) |
| (declare-fun x4 () Bool) | (declare-fun x3 () Bool) | (declare-fun x3 () Bool) |
| (assert (xor x4 x3)) | (assert (xor x3 x4)) | (assert (not (and x3 x4))) |
| (assert (and x2 x1 x3)) | (assert (and x1 x3 x2)) | (assert (let ((x5 (not x1))) x5)) |
| (assert (not (and x3 x4))) | (assert (not (and x4 x3))) | (assert (xor x3 x4)) |
| (assert (let ((x5 (not x1))) x5)) | (assert (let ((x5 (not x1))) x5)) | (assert (and x3 x2 x1)) |
| (check-sat) | (check-sat) | (check-sat) |
| (exit) | (exit) | (exit) |
Chapter 3

Normalization of SMT scripts

This chapter describes the conceptual thought process that went behind the algorithms of the SMT normalizer. The order of asserted statements and arguments, variable renaming methods and limitations are the main topics of this chapter.

3.1 Prelude

The thought process behind normalization of SMT scripts is to produce the minimal script from a given input script.

The minimal script can be described as a script whose argument variables and assertion statements are sorted in a lexicographical order, the asserted statement with the smallest lexicographical order appears first and succeeding assertions follows.

This means that the SMT normalizer should sort assertion and arguments in canonical form. A solution to this problem is to try and solve a combination of assertion statements and arguments that provides the least effort of variable renaming.

3.2 Sorting assertion statements and arguments

The sorting of assertion statements is done by recursively comparing the operators and arguments of each statement. The operators are compared as strings by their ascii value and length. As an example, the [*] operator is smaller than the [+] operator. Naturally, this means that logic operators such as [and] is larger than [*] because of the length difference. The following describes the order in which operators are sorted.

[*] < [+] < [-] < [/] < [<] < [=] < [>]

The SMT normalizer will sort assertion statements and arguments in canonical form to generate the minimal script.
The minimal script can be described as a script whose argument variables and assertion statements are sorted in a lexicographical order, the asserted statement with the smallest lexicographical order appears first and succeeding assertions follow.

Lexicographical order means that words are arranged similar to the order that they would appear in a dictionary. As an example, the word ‘‘a1a2’’ would appear before the word ‘‘b1b2’’.

The definition of lexicographical order [8] is as follow. Given two partially ordered sets \( A \) and \( B \) where \( \{a, a'\} \in A \) and \( \{b, b'\} \in B \), the lexicographical order of the cartesian product \( A \times B \) is defined as

\[
(a, b) \leq (a', b') \text{ if and only if } a \leq a' \lor (a = a' \land b \leq b')
\]

The lexicographical order can be extended to lists of arbitrary lengths by recursively applying the definition. The comparison of terms and expressions is described in further detail in the last section of this chapter. The SMT normalizer sorts assertions based on the lexicographical order of their arguments with respect to number of arguments the expression holds. As an example, the following partial script is sorted in lexicographical order.

\[
\begin{align*}
\text{(assert (and v0 v1))} \\
\text{(assert (and v0 v1))} \\
\text{(assert (and v0 v1 v1))} \\
\text{(assert (and v2 v1))}
\end{align*}
\]

However, because the SMT normalizer takes the number of arguments into account, the SMT normalizer would sort the same script as follows.

\[
\begin{align*}
\text{(assert (and v0 v1))} \\
\text{(assert (and v0 v1))} \\
\text{(assert (and v2 v1))} \\
\text{(assert (and v0 v1 v1))}
\end{align*}
\]

The result will be a script whose subsets of assertions with the same operator and number of arguments are lexicographically ordered. The reasoning behind this altered method of lexicographical ordering has to do with the implementation of the sorting algorithm that is used in the SMT normalizer and will be explained in the coming chapter.

As we can see, the alteration does effect the order of asserted statements. This is however irrelevant since the reason lexicographical order is used is to define a ordered structure on the normalized scripts.
3.3 Variable renaming

3.3.1 Prelude

The normalizer renames input variables to a standard defined range of output variables \(\{v_0, \ldots, v_n\}\) by using a hash map, so that semantically equal scripts have similar variable mapping. During the early phases of the project, the idea was that the sorting of assertions and arguments would allow the SMT normalizer to simply rename and map input variable to the corresponding output variable as it parses the script. There are however issues with this approach, one in particular is relevant to the commutative properties operators and the underlying logic. As an example, if we were to have the following scripts:

\[
\begin{align*}
\text{script T} & : \quad \text{def } x \quad \text{def } y \\
& : \quad \text{assert } (\neg x \: y) \\
\text{script U} & : \quad \text{def } y \quad \text{def } x \\
& : \quad \text{assert } (\neg x \: y)
\end{align*}
\]

We can observe that apart from the first lines of identifier declarations, the asserted expression are identical. If we were to assign output variables based on the definitions, the normalizer output for these two identical scripts would differ. Since no logic has been declared, the subtraction operator is intuitively non-commutative and does not allow us to swap arguments.

\[
\begin{align*}
\text{script T}' & : \quad \text{def } v1 \\
& : \quad \text{def } v2 \\
& : \quad \text{assert } (\neg v1 \: v2) \\
\text{script U}' & : \quad \text{def } v1 \\
& : \quad \text{def } v2 \\
& : \quad \text{assert } (\neg v2 \: v1)
\end{align*}
\]

Another approach would be to sort assertions based on the operators, number of arguments and the type of the arguments. The resulting transformation would have have assertion statements with similar structure grouped together.

This would narrow down the possible positions that an arbitrary assertion can take in the output script however, variable analysis would have to be applied on assertions that have similar structure.

3.3.2 Approaches

An approach is to model these groups of similar structure assertions as matrices, where each row represents an asserted statement within the group and each column represents the arguments for the statements. The idea is to apply a transformation on the matrix such that the resulting matrix resolves variable names and defines a order on the asserted statements.
As an example, we will assume that we have the following group of similar assertions within our script.

<table>
<thead>
<tr>
<th>Code</th>
<th>Matrix representation</th>
<th>Variable assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| (assert (and b b a d))                    | \[
\begin{pmatrix}
  b & b & a & d \\
  a & b & c & d \\
  i & j & d & l
\end{pmatrix}
\] | \[
\begin{pmatrix}
  V_0 & V_0 & V_1 & V_2 \\
  V_1 & V_0 & V_3 & V_2 \\
  V_4 & V_5 & V_2 & V_6
\end{pmatrix}
\] |
| (assert (and a b c d))                    | \[
\begin{pmatrix}
  b & b & a & d \\
  a & b & c & d \\
  i & j & d & l
\end{pmatrix}
\] | \[
\begin{pmatrix}
  V_0 & V_0 & V_1 & V_2 \\
  V_1 & V_0 & V_3 & V_2 \\
  V_4 & V_5 & V_2 & V_6
\end{pmatrix}
\] |
| (assert (and i j d l))                    | \[
\begin{pmatrix}
  b & b & a & d \\
  a & b & c & d \\
  i & j & d & l
\end{pmatrix}
\] | \[
\begin{pmatrix}
  V_0 & V_0 & V_1 & V_2 \\
  V_1 & V_0 & V_3 & V_2 \\
  V_4 & V_5 & V_2 & V_6
\end{pmatrix}
\] |
| ...                                       |                       |                     |

Researching methods of renaming variables using this model gave some interesting results. Consider a set of syntactically equal expressions as the set \( M_{m,n}(\mathbb{Z}) \) of m-by-n matrices over integers (representing variables).

We state that two matrices \( A, B \in M_{m,n}(\mathbb{Z}) \) are equivalent if \( A \) can be obtained from \( B \) using the following matrix operations.

- permuting by swapping rows.
- permuting integers by applying a bijection \( \mathbb{Z} \rightarrow \mathbb{Z} \) to every element of \( B \).

The problem of recognizing if \( A \) is equivalent to \( B \) is related to the graph isomorphism problem of determining whether two finite graphs are isomorphic.

Given a m-by-n matrix \( A \in M_{m,n}(\mathbb{Z}) \), the problem can be reduced to graph isomorphism by defining a set \( S \) to be the set of integers which appears at least once as an entry of \( A \). We can then construct a graph \( G(A) \) with \( m + mn + |S| \) labeled vertices using the following operations.

- For each row \( i \), the graph has a vertex \( u_i \) labeled as "row".
- For each cell \((i, j)\) the graph has a vertex \( v_{ij} \) labeled as "cell in column \( j \)".
- For each integer \( k \in S \) the graph has a vertex \( w_k \) labeled as "entry".
- For each cell \((i, j)\) connect \( u_i \) and \( v_{ij} \) by an edge, and connect \( v_{ij} \) and \( w_{a_{ij}} \) by an edge.

Then we can see that two matrices \( A,B \in M_{m,n}(\mathbb{Z}) \) are equivalent iff there exists an isomorphism between \( G(A) \) and \( G(B) \) which preserves labels. However, the isomorphism problem of graphs with labeled vertices is equivalent to the isomorphism problem of graphs with unlabeled vertices.

The graph isomorphism problem is a special case in computational complexity theory as it is a problem that is NP, but is neither known to be solved in deterministic polynomial time nor to be NP-complete. As of 2008, the best theoretical algorithm has a run-time complexity of \( 2^{O(\sqrt{n \log n})} \) for \( n \) vertices [5].
3.3.3 Final implementation

Since there is no known solution that can solve the problem of variable renaming faster or better than exponential time complexity, the final implementation is an algorithm that is very inefficient in terms of runtime complexity but will still provide a solution to the problem.

To ensure that the SMT normalizer will output the same script for every input that is semantically equal, the normalizer needs to define an order on the output script. The SMT normalizer will order assertions in a lexicographical order based on the mapped output variables.

Before describing the general outline for the algorithm, it is important to understand the definition of the minimal script. The minimal script is defined as the script whose arrangement of asserted expressions yields a script in which variables are sorted in a lexicographical ascending order.

The algorithm takes a list of asserted commands as input, sorts assertions and renames variables simultaneously. When a possible minimal script is computed, it is temporarily saved as the algorithm tries a different combination of assertions and variable mapping. As a slight optimization procedure, the algorithm discards partial solutions that will already prove to be worse than the best solution computed so far.

Figure 3.1 shows the operations that the SMT normalizer performs in order to compute the minimal script.
<table>
<thead>
<tr>
<th>Assertion</th>
<th>Extension</th>
<th>Sorting remaining assertions</th>
<th>Temporary solution</th>
<th>Best solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(and (a b c d))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>Null</td>
</tr>
<tr>
<td>(and (a b c d))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>Null</td>
</tr>
<tr>
<td>(and (a b c d))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>Null</td>
</tr>
<tr>
<td>(and (a b c d))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>(and (v0 v1 v2))</td>
<td>Null</td>
</tr>
</tbody>
</table>

Figure 3.1: testrun of the SMT normalizer
Figure 3.2: SMT normalizer flowchart.
Figure 3.2 gives a better overview of the algorithm, however, there are some details that need to be explained. First and foremost, by examining assertions, we are extending the variable mapping table by translating input variables to their corresponding output variables. A temporary solution saves the assertions in the order they are examined and is used to compare against the best-known solution.

The forthcoming conditional state examines if the temporary solution is worse than the best known solution. If this is the case, the normalizer removes the assertion from the temporary solution and performs a rollback operation on the mapping table in order to remove invalid variable mappings. It then tries to perform the same operation with the next entry in the list of assertions.

When the current assertion is the last to be examined, the temporary solution becomes the best known solution. If there are assertions waiting to be examined, the current assertion is removed from the list and then sorted using the extended variable mapping.

As we can see in figure 3.2, the normalizer is recursively calling the remaining assertions as it extends the variable mapping with the remaining asserted statements. It is important to note that the recursive instances have their own temporary copy of the mapping table and the list of remaining statements.

Before the recursive call, the normalizer sorts the remaining assertions using the extended variable mappings. This is also a sort of optimization procedure that could help the normalizer determine if a temporary solution is worse than the best known.

![Figure 3.3: The normalizer recursion tree.](image)

Figure 3.3 shows the recursion tree that grows as the algorithm tries to compute the minimal script. Each statement in the figure is an asserted command. As mentioned previously, the normalizer will try to compute combinations of statements and variable mappings in order to compute the minimal script. Because of the sorting of assertions before recursive calls and the comparison procedure of the temporary and best known solution, the recursion tree will only develop into a partial tree, where some branches are skipped.
Sorting the list of assertions is done by recursively comparing assertions and their respective arguments. The sorting algorithm makes use of functions that are designed to evaluate and deduce the order of asserted statements. It determines the order of assertions by examining the difference in the number of arguments within an expression. If the difference cannot be determined by the number of arguments, a deeper comparison is applied. The sorting algorithm recursively compares operators, constants and variables by using the temporary mapping table in order to determine the order. The following model illustrates the recursive behavior of the evaluation function.

![Evaluating expression](image1)

Figure 3.4: The recursive evaluation of asserted statements.

Pages 26 and 27 show detailed flowcharts of the two components. The figures with double circles are function calls from one component to the other, these components recursively call each other depending on the complexity of the evaluated expression.

The function that compares the order of assertions is also used to determine whether or not the temporary variable mappings is worse than the best-known solution. In this case, the function is used to compare the outcome of the script by comparing the same statements with different variable mappings.

### 3.4 Sorting of arguments

The sorting of arguments has been explained in the beginning of this chapter, there are however issues with the solution that has yet to be solved. First and foremost, in order to compute the minimal output of a given script, the normalizer algorithm needs to test different orders of arguments for each asserted statement in the same way it does for the order of assertions. Implementing the same algorithm for the order of arguments would however add a significant complexity overhead on the current algorithm, since the algorithm of sorting assertions is inefficient in terms of runtime complexity.
Figure 3.5: Flowchart of evaluating arguments.
Figure 3.6: Flowchart of evaluating expressions.
Chapter 4

Implementation

The next step in the development process was to design the structure of the application. Most of the structure and design of the application was already predefined by the scrambler so the correct approach is to adapt to those design guidelines when designing the normalizer.

4.1 Prelude

The normalizer is built on top of the benchmark scrambler. This proved to be a very efficient decision since the benchmark scrambler already has functionality such as a working script parser and other internal functions that the normalizer needs to utilize. The SMT normalizer is written in C++ using additional tools such as yacc [12] and bison [11] in order to parse SMT scripts and transform the script into an abstract syntax tree.

4.2 Data representation

The SMT normalizer parses the input script using yacc and bison. The result is a vector with commands. Each command is built using a predefined structure named node. The important attributes of a node is that it contains a symbol and a vector of children. An illustration that shows how expressions can be built using this data structure is shown in figure 2.1.

```cpp
std::vector<node *> commands;
```
struct node {
    std::string symbol;
    std::vector<node *> children;

    bool needs_parens;
    void add_children(std::vector<node *> *c);

    void set_parens_needed(bool b) { needs_parens = b; }
};

The SMT normalizer only operates on the vector of commands and does not alter the underlying structure of the script in order to preserve information. That is also the reasoning behind using a variable mapping table to translate input variables to their respective output variables.

typedef std::tr1::unordered_map<std::string, std::string> NameMap;
NameMap var_map;

4.3 Implementation

The normalizer takes its input from a script defined in the “SMT-LIB v2” syntax [2] and attempts to normalize the script using a set of predefined procedures. First and foremost, the normalizer sorts equation parameters in a lexicographical order. It will only then sorts equations defined within the script using the same order with respect to scope boundaries.

4.3.1 Sorting expressions

The normalizer sorts arguments and row positions of assert commands with respect to the command priority. With the help of string comparison functions in C (such as strcmp()), the arguments in an expression is sorted by type and magnitude. Arguments such as nested expressions are processed before other terms. The order is as following:

<spec_constants> := binary | hexadecimal | decimal | numeral
<s_expr> := ( <s_expr> * ) | <symbol> | <spec_constants> | <variable>

While this handles the overall sorting of similar terms and operators, a more intricate comparisons is needed for nested expressions. For example, in the case of two nested expressions with the same operator and the same number of arguments occurring, the normalizer has to order them appropriately. The order of expression is deduced by two functions that recursively calls each other while comparing two asserted expressions as shown in figure 3.4.
The functions return an enum value that determines the order of the compared assertions.

```c
#define LESS 1
#define EQUAL 2
#define GREATER 3
```

The output that these recursive functions generate depends on the comparing arguments. To understand the table below, it is important to understand that a node evaluates to NULL when it represents an expression. In that case, the child vector for that node represents the operator and arguments in the expression. Variables that are not covered in the mapping table are muted during comparison and are represented with the ‘‘”’ symbol. The ‘‘*’’ symbol represents a variable that has been translated using the mapping table and is not muted from comparison with other variables.

<table>
<thead>
<tr>
<th>arg1</th>
<th>arg2</th>
<th>operator</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>!NULL</td>
<td>N/A</td>
<td>GREATER</td>
</tr>
<tr>
<td>!NULL</td>
<td>NULL</td>
<td>N/A</td>
<td>LESS</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>&lt;</td>
<td>GREATER</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>&lt;</td>
<td>LESS</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>&gt;</td>
<td>LESS</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>&gt;</td>
<td>GREATER</td>
</tr>
</tbody>
</table>

The two functions are called ‘‘node_order’’ and ‘‘vec_node_order’’. The two functions form a mutual recursion to deduce the order of assertions. In C++, a forward declaration [6] is required to achieve mutual recursion as it is necessary to reference a function that has yet to be declared. As we can see from the code block below, a reference to ‘‘vec_node_order’’ is declared before defining the function ‘‘node_order’’.

```c
int vec_node_order(NameMap *v, NameMap *b, std::vector<node * > v1, std::vector<node * > v2);
int node_order(NameMap *v, NameMap *b, node *n1, node *n2)
{ ... }
```

The vector of commands is sorted before and during the normalization process. The sorting is achieved by using the ‘‘std::sort’’ [10] function in the C++ standard library. ‘‘std::sort’’ takes three parameters as its arguments, the first and second parameter denotes the range of elements that should be sorted in the vector. The elements are compared using the third argument as its comparison function. The comparison function takes two arguments from ‘‘std::sort’’ and returns a boolean value that determines the order of the two arguments within the vector.
std::vector<node *> sort_command_vector(std::vector<node *> *cmds, NameMap *v)
{
    /* Comparator used as the third argument to std::sort */
    std::sort((*cmds).begin(), (*cmds).end(), Comparator(*v));
    return *cmds;
}

As seen in the code block above, the third argument is a reference to a locally defined struct. The comparison function that we want to use takes more than two arguments, the additional arguments are references to the variable mapping table. In other computer languages, this could be resolved by defining a local function in which references to the variable mapping tables can be defined as a variable however, C++ does not allow locally defined functions.

This can be resolved by defining a local struct with an overloaded operator function. When we call the constructor, we pass a reference to the mapping table that we want to use during comparison of asserted expression. The overload on the ‘‘operator ()’’ function allows the struct to act as a function when the constructor is called.

struct Comparator {
    NameMap paramA;
    Comparator(NameMap paramA) { this->paramA = paramA; }
    /*
    Returns
    GREATER: true
    LESS / EQUAL: false
    */
    bool operator () (node *n1, node *n2) {
        return (node_order(&paramA, &paramA, n1, n2) <= EQUAL ) ? false : true;
    }
};
4.3.2 Variable renaming

Expression variables are stored in a mapping table of type ‘‘NameMap’’. Expression variables are never overwritten, they are rather translated. This is a good praxis as it is a non-destructive procedure which preserves information about the input script. Before the normalization procedure, the mapping table stores variable names as keys with the ‘‘_’’ symbol as their value.

This is done to denote that the variable has not yet been assigned an output variable. In order to recognize variables in the asserted expressions, the parser appends the ‘‘?’’ symbol at the beginning of the variable reference to denote the fact. The function ‘‘node_mute’’ populates this mapping table by recursively examining expression arguments. The function is used to initialize the mapping table before normalization.

```c
void node_mute(node *n, NameMap *q)
{
    if (!n->symbol.empty()) {
        if (first(n->symbol) == '?') {
            (*q)[n->symbol] = "_";
        }
    }
    for (size_t i = 0; i < n->children.size(); ++i) {
        node_mute(n->children[i], q);
    }
}
```

The mapping table is populated by examining arguments in asserted expressions. As previously mentioned, input variables are tagged with the ‘‘?’’ symbol. This makes it easier to identify variable arguments in the expression. The mapping table also stores values for a key named ‘‘MAX-KEY’’. The purpose for this key will be explained later in the chapter.

As explained in the previous chapter, the normalizer will try different assignment of variables in order to deduce the variable assignment which results in the minimal script. This means that a rollback operation needs to be performed on the mapping table. The ‘‘MAX-KEY’’ key/value is used as a pointer, denoting how many variable assignments the rollback procedure has to undo.
4.3.3 Normalizer algorithm

The normalizer algorithm consists of two functions, ‘‘normalizer’’ and the auxiliary function, ‘‘normalizer_aux’’. The ‘‘normalizer’’ performs the initial procedures before ‘‘normalizer_aux’’ performs the normalization. The initial procedures consists of initializing the mapping table as described above, sorting the vector of asserted commands by operator and argument type.

During initialization, `assert` and `declare-fun` statements are filtered out from the script and stored in separate vectors of commands. This separation is crucial since the auxiliary function only operates on assertion statements.

The ‘‘normalizer_aux’’ saves the current ‘‘MAX-KEY’’ before extending the mapping table with the current assertion that is being examined. After extension, the translated expression is stored in a temporary vector of commands. This vector is used to compare against the best-known solution and to be built upon as ‘‘normalizer_aux’’ examines the forthcoming expressions. Using the function ‘‘vec_node_order’’, the normalizer compares the partial vector to the best-known solution vector using their respective mapping table for variable translation.

If the partial solution is worse than the best-known solution according to ‘‘vec_node_order’’, a rollback operation is performed on the partial vector, using the temporary variable which stored the ‘‘MAX-KEY’’ before extending the mapping table. For each examined assertion in the loop, ‘‘normalizer_aux’’ performs a recursive procedure that copies the state of the mapping table, deletes the current assertion from the vector of commands and sorts it using the extended mapping table. After the recursive call, the current expression is removed from the partial vector and rollback operations are performed to prepare for the next expression to be examined.

The current minimal script is found when the last expression in the command vector is examined. The vector containing the minimal script is overwritten by the partial vector, a mapping table with the best-known variable assignment is also overwritten by temporary mapping table that has extended through previous iterations. As previously mentioned, ‘‘normalizer_aux’’ might find smaller scripts during the process. The ‘‘normalizer_aux’’ has found the minimal script when all combinations of assertions has been exhausted.

Combinations of assertions that will result in a script that is worse than the currently smallest script can however be foreseen and avoided. This is one of reasons why the temporary script which has been built up to that point is compared against the smallest known script.

As previously explained, the command vector is always sorted after each iteration with the help of the extended mapping table...

When ‘‘normalizer_aux’’ has found the absolute minimal script, the vector containing the best known solution and the mapping table is used to create a new vector of commands. This newly created vector will be used to merge the assert commands with the remaining script that was previously filtered out.
bool def_order(node* a, node* b)
{
    /* Sorting according to variable renames */
    return ((var_map[a->children[0]->symbol]).
        compare(var_map[b->children[0]->symbol]) < 0);
}

/* Sorts definitions lexicographically in ascending order */
void sort_definitions(std::vector<node*> *defs)
{
    std::sort((*defs).begin(), (*defs).end(), def_order);
}

With the help of the mapping table, the vector containing variable declarations are sorted lexicographically in ascending order according to the output variable name. This vector is then merged with the remaining script, overwriting previous variable declarations.

Variable names in the script are never overwritten but rather translated, the mapping table is used in conjunction with the ‘‘print_node’’ function which writes the script to standard output. Figure 4.2 illustrates the normalization process done by the ‘‘normalizer’’. When the input script is normalized, the resulting script is written to standard output.
Figure 4.2: Flowchart describing the normalization process.
Chapter 5

Evaluation

This chapter describes the final product, program execution and evaluation.

5.1 Software requirements

The software was built and implemented with the following tools and versions.

- gcc 4.7
- flex 2.5
- bison 3.0.2

5.2 Program compilation

The project includes a Makefile [13] that makes the process of building the normalizer trivial. It is crucial however, that the software requirements described in the above section is fulfilled for a successful build.

5.3 Program execution

The SMT normalizer is a terminal application that reads input files from standard input REF. The following commands issues the SMT normalizer to normalize a input script assuming the working directory contains the program and the input file `input_file.smt2`. Like most terminal applications, using the `--help` flag will give the user a list of available commands and flags that the user may utilize.

Syntax: `./normalizer < input_file.smt2`
The program will intuitively return nothing if the input script is empty. If the input script contains syntax errors, the program will halt and write the following message to standard output.

**ERROR: syntax error**

As explained above, the normalized script is written to standard output which is typically the terminal prompt for the ordinary user. Using built-in unix commands such as redirects [10], the normalized can be written to a file using the following command.

**Syntax:** 
`./normalizer < input_file.smt2 > output_file.smt2`

### 5.4 Testing

#### 5.4.1 Code refactoring legacy code

The SMT normalizer is written as an extension on the codebase for the benchmark scrambler. The benchmark scrambler however, contains functions that are meant for the specific purpose of permuting commands and command arguments. These functions had to be removed as they do not serve any purpose for the SMT normalizer.

At the time, no suitable tool could be found for checking variable dependancy between the files of the source code, so this had to be done manually. The easiest and safest method for looking after dead code is to perform a coverage test [17], together with test cases that guarantees near total code coverage. This was done with a tool that comes installed with the GCC [18] compiler, called GCOV [18]. The process was to iteratively extend the test cases while analyzing the code coverage report generated by GCOV. A tool named GCOVR [16] was used to summarize the code coverage statistics as shown in fig 5.1.

The focus of the code coverage test was to identify dead code and refactoring necessary legacy code based on branches that are never taken.

<table>
<thead>
<tr>
<th>File</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lexerc.cpp</strong></td>
<td>36.4 % 126 / 346</td>
</tr>
<tr>
<td><strong>lexerc.ly</strong></td>
<td>16.7 % 10 / 60</td>
</tr>
<tr>
<td><strong>normalizer.cpp</strong></td>
<td>90.8 % 295 / 325</td>
</tr>
<tr>
<td><strong>normalizer.h</strong></td>
<td>50.0 % 1 / 2</td>
</tr>
<tr>
<td><strong>parser.cpp</strong></td>
<td>43.4 % 98 / 226</td>
</tr>
<tr>
<td><strong>parser.ly</strong></td>
<td>28.1 % 57 / 203</td>
</tr>
</tbody>
</table>

Figure 5.1: Code coverage report of the normalizer generated by GCOV / GCOVR.
5.4.2 SMT Benchmarks

The SMT normalizer was tested with the SMT benchmarks provided by the SMT initiative. A shell script scrambles the input benchmark using the SMT benchmark scrambler with a random seed and saves the results to disk. This is done a number of times to generate multiple results of the same script. The generated results are then used as input for the SMT normalizer and the result is saved to disk.

The GNU tool diffutils [15] is used to compare the differences of the result provided by the SMT normalizer, if the SMT normalizer is working as intended then there should be no differences between the results. As explained in the previous chapters, the SMT normalizer does not handle the sorting of expression arguments under commutative operator. For testing purposes, The benchmark scrambler was modified so that the order of arguments under commutative operators are not scrambled.

As predicted before the testing phase was conducted, the SMT normalizer takes a long time calculating the normalized results of scripts containing a set of similar expressions. The runtime complexity increases drastically with the number of consecutive statements contained in a benchmark script. This is a result of the brute-force nature of the SMT normalizer algorithm.

The benchmark presented in fig 5.2 demonstrates how the number of statements in a benchmark script affects the runtime of normalizing the benchmark. As illustrated in 5.2, it takes the SMT normalizer approximately twice as long to calculate the benchmark script ‘prime_come_sat_5.smt2’ compared to ‘prime_come_sat_4.smt2’ with a difference of only 2 consecutive statements and variable declarations.

<table>
<thead>
<tr>
<th>Category</th>
<th>Problem set</th>
<th>Runtime:</th>
<th>Variable declarations:</th>
<th>statements:</th>
</tr>
</thead>
<tbody>
<tr>
<td>calypso</td>
<td>problem-001542.cvc.1.smt2</td>
<td>10 ms</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>calypso</td>
<td>problem-001563.cvc.1.smt2</td>
<td>10 ms</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>calypso</td>
<td>problem-001649.cvc.1.smt2</td>
<td>20 ms</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>calypso</td>
<td>problem-002267.cvc.1.smt2</td>
<td>&gt; 60000 ms</td>
<td>43</td>
<td>18</td>
</tr>
<tr>
<td>rings</td>
<td>all</td>
<td>10 ms</td>
<td>26 &lt; x &lt; 47</td>
<td>1 &lt; x &lt; 6</td>
</tr>
<tr>
<td>prime</td>
<td>prime_come_sat_2.smt2</td>
<td>10 ms</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>prime</td>
<td>prime_come_sat_3.smt2</td>
<td>310 ms</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>prime</td>
<td>prime_come_sat_4.smt2</td>
<td>22.940 ms</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>prime</td>
<td>prime_come_sat_5.smt2</td>
<td>45.300 ms</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 5.2: Result from running selected benchmarks.
Chapter 6

Conclusions

This chapter summarizes all the parts of the thesis work and provides recommendations for the future.

6.1 Summary

SMT solvers in the domain of software engineering are useful for verification and proving the correctness of programs. They are however, being recognized as important due to their applications in other domains such as job scheduling planning. The purpose of normalizing SMT instances in most cases is to identify equal instances in terms of semantics.

As described in the introduction, the SMT normalizer could be used as a means to reduce redundancy in caches by identifying cache lines with similar content through normalization. The process of normalizing SMT instances proved however to be a problem that suffers from complexity issues that have yet to be solved. The implementation presented in this thesis is efficient enough for solving benchmarks in the SMT-LIB library with fewer assertion statements.

It is however very inefficient for solving benchmarks with many assertion statements due to the normalization procedure of computing the minimal script. The implementation is also incomplete, the SMT normalizer does not try different arrangement of expressions and arguments in order to compute the absolute minimal script.

This means that two semantically equivalent script with a different arrangement of arguments will result in different normalization results. This was not implemented due to time constraints and the added complexity of normalization, it is however a very important part to consider. The task of normalizing SMT scripts is a problem that has to be further researched in order to provide a implementation that is efficient in terms of computational complexity.
After the research, implementation and evaluation of the SMT normalizer, answers to research questions were formulation.

- How are SMT expressions normalized?

SMT scripts are normalized by sorting argument operators under commutative operators after a defined order and representing expression with symmetrical relation properties in exactly one form. The correct arrangement on assertion declaration will provide the minimal script for the given input. This is computed by trying different sorting arrangement on asserted statements whilst renaming variable definitions.

- How are SMT definitions renamed and sorted?

As the parser reads variables, they are stored as keys inside a ordinary mapping table with empty values. The values are later computed by the normalization procedure that computes the minimal script. When the absolute minimal script has been generated, the mapping table is updated with values where each key has a value that corresponds to the renamed definition.

Instead of overwriting variable definitions, the SMT normalizer translates definitions using the mapping table when the minimal script is generated. The definitions are sorted in lexicographical ascending order based on the translated variable name.

- How are expressions sorted in the resulting transformation?

The sorting of expressions is the last operation that is applied on the SMT script. Expressions are sorted by testing different arrangement of expressions whilst renaming variables. The minimal script is defined as the script whose arrangement of asserted expressions yields a script in which variables are sorted in a lexicographical ascending order. When a minimal script is found, it is stored and compared against other possible combination candidates until the absolute minimal script is computed.
6.2 Future work

As described in the summary, the implementation that was conceived during the thesis work is incomplete and inefficient for larger SMT scripts in terms of computational complexity. A large portion of the thesis work was spent on devising a normalization algorithm that was efficiently usable for larger scripts.

Early in the project, expression arguments were sorted recursively by their respective type and value. The following describes the sorting of arguments.

\[
[s\text{-expression}] < [\text{numeral}] < [\text{decimal}] < [\text{hexadecimal}] < [\text{binary}] < [\text{variable}]
\]

By sorting the expression arguments and expressions by operator, the SMT normalizer would generate a standard form for the given input script. As an example, if we were to consider an SMT instance whose arguments are of different types we would get the following:

\[
(\text{assert} \ (\text{and} \ (\text{and} \ v0 \ v1) \ (\text{or} \ v0 \ v1) \ 1 \ 5 \ 2.0 \ \#x1A \ \#b101 \ \text{v0} \ \text{v1}))
\]

The issue that is overlooked is that it makes the process of renaming variables impractical. Since the order has to follow the

This makes the process of ordering renamed variable harder, since the arguments in the minimal script has to be sorted in a lexicographical order as well as preserving a defined order on the types of arguments. To uphold these two orders would add a complexity overhead with the current implementation.

After many setbacks, it was decided and proven that the problem of finding a efficient solution will extend the time-plan originally conceived for this thesis. The problem of normalizing SMT scripts is topic that is closely related to the problem of solving the graph isomorphism problem in mathematics.

The implementation of normalizing SMT scripts will be efficient to implement if a efficient solution for the graph isomorphism problem will ever be devised. Future researchers might also try different approaches to the problem in order to solve it more efficiently, alternatively optimizing and completing the current implementation to achieve a reasonable runtime complexity.

Another idea would be to implement a normalization procedure that would split a script into several smaller scripts, exploiting the fact that a collection of statements is sent to the SMT solver for it to solve upon each \texttt{check-sat} command. The problem is that even these scripts might not be small enough for the normalizer to solve in a reasonable time.
References


