Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 62

Model Counting of String Constraints for Probabilistic Symbolic Execution

Yannic Noller

Course of Study:	Softwaretechnik
Examiner:	Dr. rer. nat. Antonio Filieri
Supervisor:	Dr. rer. nat. Antonio Filieri
Commenced:	2015/10/28
Completed:	2016/04/15
CR-Classification:	D.2.4, D.2.5, F.4.1

Abstract

Probabilistic symbolic execution is a static analysis technique aiming at quantifying the probability of a target event occurring during a program execution; it exploits symbolic execution to identify the conditions on the program inputs leading to the occurrence of the target event and then model counting to quantify their probability. While efficient methods exist for model counting of numeric constraints, only limited results have been obtained for counting string constraints. The constraints are indeed first encoded in a corresponding accepting automaton; then the number of accepting paths of the automaton is quantified via a convenient generating function. The encoding in the form of automata limits the expressiveness of the formalism used for constraint specification to constructs mappable into automata, for an exact model count. Furthermore, the encoding of disjunctions requires the parallel composition of the automata representing the disjuncts, leading to an exponential growth in the size of the resulting automata.

This thesis introduces the usage of SMT solvers to count the models of a string constraint by leveraging a standard smtlib interface. Several algorithms for both exact and approximate model counting of string constraints are defined and compared. The different solutions are implemented on top of the SMT solver CVC4 and evaluated on a set of established benchmarks, demonstrating the increased expressiveness with respect to previous approaches and improved performance on several classes of problems.

Zusammenfassung

Probabilistic Symbolic Execution ist eine statische Analysetechnik, um die Eintrittswahrscheinlichkeit eines Ereignisses in der Programmausführung zu bestimmen. Es verwendet dazu eine Kombination der herkömmlichen Symbolic Execution und dem Zählen von Modellen. Symbolic Execution extrahiert die Pfadbedingungen, die notwendig sind, um zu dem Ereignis zu gelangen. Indem man die Anzahl Möglichkeiten berechnet, diese Bedingungen zu erfüllen, kann die Eintrittswahrscheinlichkeit des Ereignisses bestimmt werden. Während für das Zählen von Modellen für numerische Bedingungen schon effiziente Methoden publiziert wurden, existieren für das Zählen von Bedingungen mit Zeichenketten zurzeit nur eingeschränkte Möglichkeiten. Die existierenden Techniken basieren auf der Konstruktion von endlichen Zustandsautomaten, deren Sprache die erfüllenden Zeichenketten akzeptieren. Die Anzahl Modelle ergibt sich durch die Bestimmung der Anzahl akzeptierender Pfade in diesen Automaten. Diese Reduktion des Problems schränkt die Teilmenge an formulierbaren Bedingungen ein, für welche die exakte Anzahl Modelle bestimmt werden kann. Des Weiteren führen komplexe Bedingungen mit Verkettung von Disjunktionen zu einer Zustandsexplosion im konstruierten Automaten.

Diese Thesis stellt ein Verfahren vor, das mit Hilfe von sogenannten SMT Solvern, die Anzahl Modelle von Bedingungen mit Zeichenketten bestimmen kann. Es wurden mehrere Algorithmen entwickelt, die unter Verwendung einer standardisierten SMT Solver Schnittstelle die exakten also auch approximativen Mengen an Modellen berechnen können. Als Implementierungsgrundlage wurde der SMT Solver CVC4 verwendet. Die Evaluation des entwickelten Forschungsprototyps mit etablierten Benchmarktests zeigt die erweiterte Ausdrucksfähigkeit der SMT-basierenden Verfahren gegenüber den existierenden Ansätzen. Außerdem können bestimmte Problemklassen nun effizienter gelöst werden.

Contents

Chapter 1

Introduction

1.1 Model Counting for String Constraints

Probabilistic symbolic execution (PSE) [Geldenhuys et al., 2012] is a recent static analysis technique for reasoning about quantitative properties of a program, such as its reliability [Filieri et al., 2013]. In general, PSE can be used to quantify the probability of a target event occurring during a program execution. PSE relies on the combination of conventional symbolic execution [King, 1976] and solution space quantification techniques. Symbolic execution is used to identify the path conditions, i.e., a set of constraints on the program input, whose satisfaction by input values would determine the target event to manifest during the execution of the program. In finite domains the quantification can be achieved via model counting. Model counting techniques are used to determine the number of solutions (models) satisfying a given constraint. When the number of inputs satisfying a given path condition is computed, the probability of an input triggering the target event can be computed too, possibly taking into account the expected distributions of the inputs [Filieri et al., 2013].

While efficient methods exist for model counting of numerical constraints (e.g., `LattE` [Loera et al., 2004] for linear integer constraints and `qCoral` [Borges et al., 2014] for floating point constraints), only limited results have been obtained for counting the models of string constraints. The state of the art for model counting of string constraints is based on the construction of finite state automata (FSA) that accept the language defined by the constraints (e.g., `ABC` [Aydin et al., 2015]) and then counting the number of accepting paths of the automaton, which corresponds to the number of solutions of the constraints. Since an FSA can only represent constraints that may reduced to regular expressions, this model counting approach provides limited expressiveness. A broader classes of constraints can be handled by counting the models of a relaxation of non-regular constraints (which can be accepted by FSA); however, the result would in this case be an upperbound on the number of solutions, which may not fit for several analysis applications. Furthermore, the need to reduce the constraints to a corresponding accepting FSA limits the possibility of combining string constraints with other useful commonly used for program analysis, e.g., bounding the length of a string with the result of an integer operation. Finally, constraints with complex combinations of disjunctions and string concatenations lead to an exponential growth in the size of the resulting automata [Aydin et al., 2015], which in turn lead to a

combinatorial explosion of the number of paths and a may make the counting prohibitive.

Besides extending the applicability of PSE, model counting of string constraints has several applications, especially in security (e.g., Luu et al. [2014]). For example model counting can be used to quantify the strength of a secure password or the probability of information leakage from program execution.

Model counting problems for propositional boolean constraints, #SAT, have been proposed in literature (e.g., extending DPLL solution algorithms as in [Gomes et al., 2009]). A similar thread of research is investigating the possibility of counting the models of SMT constraints, #SMT (e.g., Chistikov et al. [2015]). The definition of SMT theories for the satisfiability of string constraints is paving a new way for the application of #SMT results to model counting problems for string constraints. These approaches have the potential of overcoming the limitations of current model counting approaches, both by extending their expressiveness beyond the constraints mappable to FSA and by allowing the combination of multiple theories within the same constraint.

1.2 Goals

The goal of this thesis is the exploration and evaluation of #SMT procedures for model counting for string constraints. A prototypical implementation of the studied #SMT procedures has also been developed to evaluate the comparison of the different approaches on a set of established benchmark.

1.3 Document Structure

The remaining thesis is structured in two parts: the first part includes Chapters 2, and 3, covering the theoretical background, the description of the investigated model counting approaches from literature, and the definition of specialized variations for counting the models of string constraints; the second part includes Chapters 4, 5, and 6, discussing the practical implementation, the evaluation of the different approaches, and the conclusion.

Chapter 2 *Background* contains the necessary background on probabilistic symbolic execution and surveys the state of the art of model counting and string constraint solving.

Chapter 3 *Model Counting with SMT Solvers* defines the proposed approaches and theories for model counting of string constraints using SMT solvers.

Chapter 4 *Implementation* describes the implementations details of all the approaches, including the design decision that have been made.

Chapter 5 *Evaluation* evaluates and compares the different approaches, identifying strengths and limitations of each of them.

Chapter 6 *Conclusions and Future Work* presents some concluding remarks and sketches the directions for future research.

Chapter 2

Background

This chapter starts with an overview about (probabilistic) symbolic execution, which represents the main application scope of model counting. The subsequent part reviews the state of the art of model counting and focuses on its application to string constraints. Additionally, the chapter includes some benchmarks that are commonly used in the area of model counting of string constraints.

2.1 (Probabilistic) Symbolic Execution

Symbolic execution (SE) was first introduced by King [1976] and means the execution of the program with symbolic instead of concrete inputs. Instead of having one execution path, SE generates an execution tree that contains different execution paths. It introduces a *path condition* (PC) for every branching point in the program e.g. if-statements and loop conditions. By solving these path conditions it is possible to generate test inputs to meet certain coverage criteria like path coverage. Symbolic execution builds the PCs incrementally during its execution and tries to avoid paths that are not reachable, i.e. paths with an unsatisfiable path condition. SE uses an SMT solver to solve the PCs.

One of the main reasons behind the success of symbolic execution is that in the last decade a dramatic increase took place in the computational power of modern computers. Efficient decision procedures were developed by using this new achievements (see Orso and Rothermel [2014]).

Although symbolic execution had a great success, it still has some problems that need to be solved. Constraint solving can be very difficult for complex constraints like non-linear constraints or constraints with native calls, which cannot be solved by SMT solvers. Therefore *dynamic symbolic execution* techniques were developed to solve such constraints by using a combination of symbolic and concrete execution (see e.g. *directed automated random testing* (DART) by Godefroid et al. [2005]). Another problem is the path explosion for e.g. loops or recursive function calls, which can be tackled with *bounded symbolic execution* techniques with a bound on the search depth.

Probabilistic symbolic execution (PSE) by Geldenhuys et al. [2012] is an extension of the classical SE by assigning probabilities to the program paths. This additional information enables the probabilistic analysis of programs and can, for example, be used to find bugs by using the most likely or unlikely path, to calculate code coverage probabilities for test

cases and to calculate the probability of bugs. PSE can also be used for a reliability analysis by using the probability of successful paths (see Filieri et al. [2013]). Geldenhuys et al. stated that probabilistic symbolic execution can be performed by a combination of symbolic execution and model counting. The following equation shows how the probability of a path constraint can be calculated:

$$Pr(c) := \frac{\#c}{\#D} \tag{2.1}$$

where $\#c$ represents the number of solutions that satisfy the constraint c and $\#D$ represents the size of the domain (which is assumed to be greater than zero).

2.2 Model Counting

Model counting means to determine the number of solutions for a logical formula. Gomes et al. [2009] describe multiple approaches for model counting that can be divided into two groups: *exact* and *approximate* model counting. Based on Gomes et al. these approaches are presented in the following sections.

Exact model counting

The earliest practical approach for model counting was based on *DPLL*-style exhaustive search. DPLL denotes *Davis–Putnam–Logemann–Loveland* and the DPLL-algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulas in conjunctive normal form (CNF), i.e. for solving the CNF-SAT problem. The model counter `CDP` by Birnbaum and Lozinskii [1999] computes the model count for a n-variable formula by exploring the complete search tree. As in the backtracking-based DPLL search, variables are selected iteratively and unsatisfiable branches are pruned. While DPLL search only tries to find out whether the formula is satisfiable, `CDP` tries to find the exact count of models for the given formula. Algorithm 1 is the recursive version of the `CDP` algorithm to count the models for a given formula F: If the current formula contains the empty clause, then it is unsatisfiable and the model count is zero. If all clauses are satisfied in the current formula, regardless from the number of assigned variables, then it returns the number 2^{n-t}, i.e. the number of all possible assignments to the unassigned boolean variables. The function returns the sum of the recursive calls, in which a selected variable was assigned to `TRUE` and `FALSE`.

The DPLL approach can also be used to calculate partial counts, i.e. upper or lower bounds. Gomes et al. mention several optimization approaches. For example, the disjoint components in F can be identified by using component analysis. These components can be solved separately and the results can be merged by multiplication. This approach is implemented in the model counter `Relsat` by Bayardo and Pehoushek [2000]. Another optimization is the caching of results for reappearing sub formulas, e.g. the research by Bacchus et al. [2003].

```
Input  : A CNF formula F over n variables; the recursion depth t initially set to 0
Output : #F, the model count of F
begin
    UnitPropagate(F);
    if F has an empty clause then return 0;
    if all clauses of F are satisfied then return 2^(n−t);
    x ← SelectBranchVariable(F);
    return CDP(F|_x, t + 1) + CDP(F|_¬x, t + 1);
end
```

Algorithm 1: Exact model counting with model counter `CDP`, called as CDP(F, 0) [Birnbaum and Lozinskii, 1999]

Another approach for exact model counting is called *Knowledge Compilation* and means the *compilation/conversion* of the given CNF formula into another logical form. By using resources and effort for the conversion, this new form is meant to provide some advantage in terms of retrieving the model count, i.e. it is capable of deducing the model count more efficient than the original form. For example Darwiche [2004] introduced the compiler `c2d`, which converts a given CNF into a form that is a strict superset of ordered binary decision trees (BDDs) called d-DNNF. The properties of this form encourage model counting by performing a topological traversal of the underlying acyclic graph.

Approximate model counting

Based on the fact that most model counting methods handle a problem within combinatorial search space, they might not scale up to larger problem sizes. Therefore, it might be easier and more suitable to provide lower or upper bounds for the model count. This is called *approximate model counting*. Gomes et al. mention that many applications of model counting do not even care about an exact count, as long as they get a sort of quality factor for the estimation. Thus one can divide approximate model counting methods into two groups: estimations *without guarantees* and *with guarantees*.

Approximate model counting without guarantees: Wei and Selman [2005] introduced their model counter `ApproxCount` that uses local search with Markov Chain Monte Carlo (MCMC) sampling to compute an approximate number of models for a given formula. The basic idea originate from Jerrum et al. [1986]. They reduce model counting for F to a simpler formula. Assume one could sample the satisfying assignments (near-)uniformly. Then select a variable x and look for the truth value that occurs more often in the sample data, e.g. this would be `TRUE`. Then build the simpler formula $F^+ = F|_{x=\texttt{TRUE}}$. This process can be repeated recursively until the resulting simplified formula can be counted by exact

model counting methods. For the complete sample data we can count the number of satisfying assignments devoted as M and we can count the satisfying assignments where x is set `TRUE` devoted as M^+. Based on the fraction $\gamma = \frac{M^+}{M}$, the model count of F is (approximately) $\frac{1}{\gamma} \times F^+$ (assuming $\gamma \neq 0$). The result of this method depends on how good one can sample (near-)uniformly. The method does not provide any guarantee for the model count. According to Gomes et al. `ApproxCount` is, compared to exact model counters, extremely fast and has been shown to provide very good estimates.

Gogate and Dechter [2007] introduced their model counter `SampleMinisat`, which uses DPLL-based SAT solvers to build backtrack-free search spaces, i.e. without unsatisfiable branches, completely or approximately. Their approach uses the importance sampling technique by Rubinstein [1981]. According to Gomes et al. `SampleMinisat` can provide very good estimates of the model count when the formula is within the reach of DPLL-based methods.

Approximate model counting with guarantees: Based on the presented model counter `ApproxCount` Gomes et al. [2007] developed the model counter `SampleCount`, which uses sampling with a modified and randomized strategy. This model counter provides provable lower bounds on the total model count with high confidence (probabilistic) correctness guarantees. The underlying sampler is not used to select the variable and to compute the multiplier γ, but to determine the order of how to select the variables. The goal is to select variables so that the search space is divided most evenly.

Another, totally different, method for model counting is called *XOR-streamlining* and was presented by Gomes et al. [2006] with the model counter `MBound`. It adds non-redundant constraints to the original problem to focus the search on a subspace. This technique called *streamlining constraints* was shown to be successful in solving very hard combinatorial design problems. By hoping that the narrowed problem still contains solutions, it might be solved much easier than the original one. `MBound` works as follows: Like the name suggests, *XOR* constraints are added repeatedly to the formula as additional CNF clauses. The constraints are chosen randomly. This extended formula is checked with a state-of-the-art SAT solver for satisfiability. Intuitively, every random XOR constraint cuts the solution space approximately half. So if the extended formula is still satisfiable and s XOR constraints were added, then the original formula must have at least of the order of 2^s models. By repeating the addition of s constraints in t experiments one can show that there is probabilistic correctness guarantee of $1 - 2^{-\alpha t}$ that the original formula has at least $2^{s-\alpha}$ satisfying assignments for any $\alpha > 0$. There are approaches to increase the efficiency by e.g. using an exact model counter instead of the SAT solver.

2.3 Solving String Constraints

The presented model counting techniques in the previous chapter 2.2 were developed for propositional formulas and the ideas cannot be mapped easily to string constraints. For

example the model counter SampleCount selects variables to assign a value of the alphabet (here TRUE or FALSE) to them. Selecting a variable in string constraints would mean selecting a character at a position in the string. This can be done, but then the string represents an AND combination of all characters. This means that it is necessary to select characters for all positions before stopping the selection and before be able to check the satisfiability of the generated string. In propositional formulas it is possible to simplify the constraint to a CNF. In order to evaluate the CNF to TRUE, it is sufficient to find an assignment for a subset of the variables that satisfies all clauses. Then the model count for this path can be calculated via $2^{number\ of\ not\ assigned\ variables}$. Assuming that we can evaluate the formula to TRUE without selecting many variables, this is very time efficient. This method does not work for string variables. Hence, there is a need for other methods and technologies that are described in the following section.

2.3.1 Automata-Based Counting

Yu and Cova [2008] proposed a new automata-based algorithm to do string analysis on PHP programs to identify security vulnerabilities. Their work can be divided into two parts: They developed a front-end that translates the PHP program into a flow graph. This flow graph is an abstract representation of the program to be able to do programming language independent string manipulations. Secondly, they developed an algorithm to analyze and transform the flow graph into a finite state automaton (FSA). The FSA is updated according to the string manipulation along the control flow, by accepting the regular language that summarizes all possible values of the string expression. At the end they generate for each string expression the intersection between the accepted strings, i.e. the built FSA, and the set of attacking strings. An empty intersection means that the PHP program is proven to be vulnerable free, otherwise the algorithm terminates with a violating counter example. Yu et al. implemented a prototype tool called STRing AutomatoN GEneratoR (STRANGER) and showed how to build an FSA to accept string expressions. That supported other researchers in that area, for example Aydin et al. [2015].

Luu et al. [2014] introduced, based on the work by Flajolet and Sedgewick [2009] on analytic combinatorics, the idea to leverage *generating functions* for model counting of string constraints. Their tool String Model Counter (SMC) transforms the given constraint into its own constraint language and produces a generating function. This function can be used to calculate the number of satisfying string values. The SMC constraint language is expressive enough to model constraints arising in real world JavaScript applications and UNIX C utilities. Luu et al. evaluated their results on an application to assess the strength of a new password in case the old password was revealed by an attacker and the attacker knows the constraints for the new password.

Similarly, Aydin et al. [2015] recently presented their automata-based model counter for string constraints called ABC. Their technique consists of two steps: (1) Construction of an automaton for accepting the satisfying values for the given variable and string constraint. (2) Given the automaton they generate a function that computes the number of

accepting values based on the length bound. Their constraint language can handle regular language queries, word equation with concatenation and replacement as well as arithmetic constraints on string length (see Table 5.4). ABC is at the moment the most efficient and most applicable model counter of string constraints. Hence, it is used as comparison in chapter 5.

2.3.2 SMT Solvers

Satisfiability Modulo Theories (SMT) problems are a decision problems for logical formulas with respect to combinations of background theories expressed in first-order logic. Intuitively SMT solvers solve SAT problems for logical formulas in predicate logic. These predicates can represent boolean variables but also variables of the mentioned background theories. Background theories can be e.g. the theory of integers, the theory of real numbers and the theory of strings.

Liang et al. [2014] introduced their technique for solving mixed theory constraints including strings and linear integer arithmetics. Their approach can be integrated into the *DPLL(T)* framework Nieuwenhuis et al. [2006] that combines an SAT solver with multiple specialized theory solvers for conjunctions of constraints over a certain theory, here linear integer arithmetics, strings and regular languages. They support a constraint language that includes quantifier-free constraints over unbounded strings with length and regular language membership. Liang et al. use an already existent, standard solver for linear integer arithmetic constraints and a new solver for string and regular language constraints. By applying various abstract derivation rules to the initial constraint inputs they check the consistency (and hence the satisfiability) of the constraints. The solvers are combined via the Nelson-Oppen theory combination [Nelson and Oppen, 1979], where entailed equalities are communicated between these solvers, i.e. here terms about the length of string variables generated via rules that are mentioned above. Liang et al. implemented the approach within their SMT solver CVC4. They did an experimental comparison with the string solvers Z3-STR [Zheng et al., 2013] and Kaluza [Saxena et al., 2010] showing that their solver is highly competitive with them.

Phan and Malacaria [2015] investigated SMT procedures for model counting with respect to a set of boolean variables. They proposed the idea to use *Blocking Clause* methods and a *Depth-First-Search* for model counting. Phan et al. built their prototype aZ3 on top of the SMT solver Z3. They showcased their solver with the *Quantitative Information Flow* analysis (see Phan and Malacaria [2014]).

2.3.3 Open Challenges

After the preliminary research on model counting of string constraints and to the best of the thesis author's knowledge there are no approaches besides the ones in the above presented sections. Therefore, state of the art approaches for model counting of string constraints use a reduction to other decision problems like the extraction of the generating function

for an automaton. This includes drawbacks like the resource-intensive construction of the automaton, which might lead to serious performance bottlenecks. More specifically, automata-based approaches have the restriction to work only on linear constraints and solving on linear arithmetics is not possible or at least not efficient. The construction of an automaton for disjunctions leads to an exponential growth in state space, which is also called state explosion. Automata-based approaches can be used *only* on string constraints and do not provide the opportunity to mix different kinds of constraints, e.g. string constraints and numeric constraints. A more general approach is needed to cover all conditions in a program.

Chapter 3

Model Counting with SMT Solvers

This chapter describes the various possibilities on how to count models with SMT solvers, also denoted as #SMT. It is divided in two parts: section 3.1 contains the general discussion of the different approaches and section 3.2 contains the the pseudo code and the discussion of various algorithms that are able to count models using the SMT API functions.

3.1 Survey of Approaches

This section contains the description of the three approaches on how an SMT solver can be used for model counting: *(i)* adjusting the string model generation of the SMT solver, *(ii)* using the SMT API and *(iii)* use constraint files as external input for the SMT solver.

3.1.1 SMT Model Generation

An SMT solver has the functionality to produce a model for a satisfiable formula. The Blocking Cause Method (see chapter 3.2.2) uses this functionality, but in quite inefficient manner like later described. In order to use this functionality in an efficient way, it is necessary to adjust the generation of the models directly on the search tree of the SMT solver. Unfortunately, this part of the solver is not well documented (e.g. for `Z3-STR` [Zheng et al., 2013] and `CVC4` [Barrett et al., 2011; Liang et al., 2014]), so it would be very cumbersome to follow this approach.

3.1.2 SMT API Usage

An SMT solver provides the API functions presented in Table 3.1 to use it for the incremental solving of constraints. These operations can be used to assert a value for the string variable for that the models should be counted. Then the formula can be checked whether it is still satisfiable and afterwards the original formula can be restored via the stack mechanism. It is also possible to incrementally check a string value for satisfiability. This means that the SMT solver acts like a *oracle* in the search for string models.

Table 3.1. API of SMT solver.

Operation	Description
Push()	Create a backtracking point.
Pop()	Backtrack to the previous point.
Assert(f)	Assert formula f into the solver.
Check()	Check satisfiability of the asserted formula.
Model(var)	Get model of the last check for variable *var*.

3.1.3 Constraint File Adjustment

In case an SMT solver does not provide the above presented API or the API is not accessible (e.g. the API support is missing for a certain programming language), then it is also possible to add the assertion that would have been made via the *Assert()* function, directly to the constraint file. Obviously, this leads to an immense overhead of I/O operations.

3.1.4 Selection of the Research Approach

Due to fact that the SMT Model Generation is not well documented and the Constraint File Adjustment leads to an immense overhead of I/O operations, this thesis focuses on model counting of string constraints with the SMT API Usage. Nevertheless at the beginning of this thesis the Constraint File Adjustment was used to check the feasibility of #SMT because the APIs of the SMT solvers were not fully accessible. The following section 3.2 provides the corresponding algorithms that search string models by using the an SMT solver as *oracle*.

3.2 Survey of Solution Algorithms

This section contains several algorithms that were identified during the research on how to use the API of an SMT solver for model counting of string constraints with a fixed length bound. They can be summarized in four categories: the **Random Search Method** (see chapter 3.2.1) that simply generates randomly string values and check the satisfiability for them, the **Blocking Clause Method** (see chapter 3.2.2) that uses the functionality of an SMT solver to produce models, **Depth-first Search Methods** (see chapters 3.2.3, 3.2.4 and 3.2.5) that systematically investigate the search tree and **Monte Carlo Methods** (see chapter 3.2.6) that kind of use a directed search. All of them can be executed in the environment algorithm, shown in Algorithm 2, which does a precheck of the constraint satisfiability and the satisfiability of the given length bound.

```
Input  : the formula F; the lengthBound for the string models
Output : #F, the model count of F
begin
    Assert(F);
    Assert(length(x) ⩽ lengthBound);
    if (Check() ≠ SAT) then
        error("the constraint is not satisfiable");
        return 0;
    end

    count = RandomSearch(..) / BlockCount(..) / DFS(..) / MCTS(..) / ... ;

    return count;
end
```

Algorithm 2: Environment Algorithm

As already mentioned, these algorithms have different characteristics that leads to that there are constraints for which one of them is more suitable than the others. The discussion of this characteristics is summarized in the chapter 5.3 *Comparison of Solution Algorithms*.

3.2.1 Random Search

The Random Search algorithm (see Algorithm 3) is meant to be a baseline for the performance of the other algorithms. It generates randomly string values and checks their satisfiability for the given variable and formula. It is the most naive approach to search for string models of a constraint. As input it requires the calculation budget, e.g. a time bound, the string variable for which the models are counted and the length bound of the string variable. As output it produces a lower bound for the model count of the given formula. Since this approach is very naive and is no systematic search, there are no checks whether there are remaining models that were not checked yet. So in principle, the algorithm runs forever, up to the point there is no remaining budget.

The function *randomChooseLength(lengthBound)* chooses the length for current iteration with the probability:

$$p(length) := \frac{|A|^{length}}{\sum_{i=0}^{lengthBound} |A|^i} \tag{3.1}$$

where A is the alphabet. Therefore the length with the most possible string values is the most likely one to be chosen. The function *randomChooseAlphabetCharacter()* chooses uniformly distributed a character from the alphabet A. In this thesis Random Search is only used with a time bound to get an impression how the other algorithms perform,

but it is also possible to use Random Search as approximate model counting algorithm. Therefore you have to calculate the necessary number of tested string values to get a statistical significant result or to assess statistical significance afterwards.

```
Input  : the calculation budget; the string variable x for which the models are
         counted; the lengthBound
Output: #F, the model count of F
begin
    count := 0;
    while (enough remaining budget) do
        length := randomChooseLength(lengthBound);

        value := "";
        for (i=0; i<length; i++) do
            value += randomChooseAlphabetCharacter();
        end

        if (value not already checked) then
            Push();
            Assert(x = value);
            if (Check() = SAT) then
                count++;
            end
            Pop();
        end
    end
    return count;
end
```

Algorithm 3: Random Search, called as *RandomSearch*(budget, x, lengthBound)

Darbon et al. [2006] show that the sample size can be calculated as:

$$N := \frac{ln(\frac{2}{\delta})}{2\epsilon^2} \tag{3.2}$$

in order to get an estimated model count for the probability equation:

$$Pr(|estimate - actualValue| \leqslant \epsilon) \geqslant 1 - \delta \tag{3.3}$$

where δ is called the *confidence* parameter and ϵ is called the *approximation* parameter. They use an simplified version of the Chernoff-Hoeffding bounds [Hoeffding, 1963].

One can assess the statistical significance of the approximate model count by using the

idea of "hit-or-miss" (HM-MC) (see e.g. Robert and Casella [2005]). The variance of the ratio between the number of satisfiable string values and the total number of generated string values can be used as accuracy measurement. As shown in Equation 2 of Borges et al. [2015] the variance decreases with the number of samples and this increases the accuracy.

3.2.2 Blocking Clause Algorithm

The Blocking Clause algorithm (see Algorithm 4) exploits the functionality of the SMT solver to produce models. The idea is to iteratively ask for a model and then add it negated to the actual formula. This avoids to produce a model twice. As input it requires the string variable for which the models are counted. An optional input is the calculation budget, e.g. a time bound. Without a calculation budget the Blocking Clause produces the exact model count. With a calculation budget the Blocking Clause produces a lower bound of the model count (or the exact count if the budget is large enough). Since it adds more constraints continuously, the solving time for the next model increases with each step.

```
Input  : the calculation budget; the string variable x for which the models are
         counted
Output : #F, the model count of F
begin
    count := 0;
    while (Check() = SAT & enough remaining budget) do
        count++;
        m := Model(x);
        Assert(x≠m);
    end
    return count;
end
```

Algorithm 4: Blocking Clause, called as *BlockingClause*(budget, x)

The algorithm does not use the API operations ***Push***() and ***Pop***() because it iteratively builds a new constraint and never needs to backtrack to a previous state of the asserted formula.

3.2.3 Depth-First Search Algorithms

The standard Depth-First Search algorithm (see Algorithm 5) investigates systematically the total search space. It iterates the complete alphabet for every position in the string value. Therefore the algorithm uses the well known recursive depth-first search with backtracking. Before investigating a path, the algorithm checks whether the current prefix of

the string value is satisfiable. This enables the efficient pruning of the unsatisfiable search space. The algorithm requires as input the current recursion depth (starting with 0), the maximum recursion depth (= length bound), the current string value (starting with the empty string) and the string variable for which the models are counted. An optional input is the calculation budget, e.g. a time bound. Without a calculation budget the DFS produces the exact model count. With a calculation budget the DFS produces a lower bound of the model count (or the exact count if the budget is large enough).

```
Input  : the current recursion depth; the maximum recursion depth (= length
         bound) maxDepth; the calculation budget; the current string value toCheck;
         the string variable x for which the models are counted
Output : #F, the model count of F
begin
    if (budget exhausted) then
        return 0;
    end

    count := 0;
    Push();
    Assert(prefix(x) = toCheck);
    if (Check() ≠ SAT) then
        Pop();
        return 0;
    end
    Assert(x = toCheck);
    if (Check() = SAT) then
        count++;
    end
    Pop();

    if (depth < maxDepth) then
        for (letter in alphabet) do
            count += DFS(depth+1, maxDepth, budget, toCheck+letter, x);
        end
    end
    return count;
end
```

Algorithm 5: Depth-First Search, called as *DFS*(0, lengthBound, budget, "", x)

To check the satisfiability the presented pseudo code always uses a full variable check,

e.g. ***Assert***(x = "pass123") or ***Assert***(*prefix*(x) = "code"), and the ***Push***() and ***Pop***() operations to reset the solver's formula. There are also other ways to do that, e.g. asserting characters for positions with a *CharAt()* operation, but this would assume that all SMT solver support this string operation. That is why the presented pseudo code tries to stay general. Chapter 4 discusses various implementation variants.

3.2.4 Randomized DFS

Since the standard DFS algorithm performs an exhaustive search and the scalability of DFS is limited (see chapter 5.2.1), it is worthwhile to investigate methods that search only in a subset of the search space. One way is to randomize the DFS on various parts. The randomization results in an estimated model count that is sufficient for many practical cases (see chapter 2.2). The following three sections show approaches on the randomization of backtracking, extracted and adjusted from Parízek and Lhoták [2011] and Lynce and Marques-Silva [2007]. These three approaches resulted in two algorithms that were implemented for this thesis. They are described in the chapter 4 *Implementation*.

Randomized Alphabet

The simplest randomization approach for DFS is to randomize the order of the traversed alphabet for every for loop (see Algorithm 6). This will make no difference for an exhaustive search without a budget, but with a budget the DFS with randomized alphabet will find on average more models. The alphabet get randomized uniformly, so there is no need for an extra parameter.

```
...
if (depth < maxDepth) then
    for (letter in randomize(alphabet)) do
        count += DFS(depth+1, maxDepth, budget, toCheck+letter, x);
    end
end
...
```

Algorithm 6: Extension of the Depth-First Search algorithm for the randomized alphabet.

Randomized Character Selection

A way to reduce the search space is to randomly skip characters. An efficient way to do that is to skip letters with a given probability and continue with the next one (see Algorithm 7). Therefore the standard DFS needs as additional input the probability s for that a character is skipped. The resulting model count is a lower bound approximation.

```
Input: ...; factor s that is used to skip a character
...
for (letter in alphabet) do
    if (random() < s) then
        continue;
    end
    count += DFS(depth+1, maxDepth, budget, toCheck+letter, x);
end
...
```

Algorithm 7: Extension of the Depth-First Search algorithm for skipping characters randomly.

Randomized Backtracking

Another way to reduce the search space is to randomly extend the step size for backtracking. This can be done by backtracking again after a previous backtrack step with a given probability (see Algorithm 8). Therefore the standard DFS needs as additional input the probability b for that the backtracking size is extended. The resulting model count is a lower bound approximation.

```
Input: ...; factor b that is used to backtrack randomly more than one step
...
for (letter in alphabet) do
    count += DFS(depth+1, maxDepth, budget, toCheck+letter, x);
    if (random() < b) then
        break;
    end
end
...
```

Algorithm 8: Extension of the Depth-First Search algorithm for backtracking randomly more than one step.

Nevertheless, the above presented randomization techniques do not lead to a uniformed sampling of the search space because shorter strings are generated with a higher probability than longer ones. Therefore it is not possible to guarantee the resulting estimated model count. The next section *DFS with Uniform Sampling* tackles this problem.

3.2.5 DFS with Uniform Sampling

In order to achieve an uniform sampling, and therefore be able to guarantee the estimated model count, the Algorithm 9 first generates the set of feasible string values, i.e. string values that are satisfiable as prefix of the given string variable, and then checks only a subset of it for satisfiability. The algorithm requires as input the length bound of the string variable, the string variable for which the models are counted and the sampling ratio. An optional input is the calculation budget, e.g. a time bound. Without a calculation budget and a sampling ratio equals to 1.0 the DFS with Uniform Sampling produces the exact model count. With a calculation budget or a sampling ration other than 1.0 the DFS with Uniform Sampling produces an approximated model count that can be lower, higher or equal the actual model count. Similar as described in chapter 3.2.1 *Random Search* the sample size and hence the sample ratio can be calculated for statistical significance.

```
Input  : the lengthBound; the calculation budget; the string variable x for which
         the models are counted; the sampling ratio
Output : #F, the model count of F
begin
    feasibleStrings := calculateFeasibleStrings(0, lengthBound, budget, "", x);
    sampleSet := calculateSubSet(feasibleStrings, ratio);
    count := 0;
    for (value in sampleSet) do
        Push();
        Assert(x = value);
        if (Check() = SAT) then
            count++;
        end
        Pop();
    end
    return count/|sampleSet| × |feasibleStrings|;
end
```

Algorithm 9: Depth-First Search with uniformed Sampling, called as *DFS_sampled*(lengthBound, budget, x, ratio)

The function *calculateFeasibleStrings(..)* performs a standard depth-first search and returns all string values that are visited during the execution and that are satisfiable as the prefix of the given string variable. This set of strings is called *feasible strings*. The function *calculateSubSet(..)* samples the feasible strings according to the given sampling ratio uniformly.

At the end the approximated count is calculated via the formula:

$$approxCount := \frac{|sat\ elements\ of\ sampleSet|}{|sampleSet|} \times |feasibleStrings| \tag{3.4}$$

3.2.6 Monte Carlo Tree Search

All the above presented algorithms lack in one characteristic: they do not search in certain direction for satisfiable strings, but they do a systematical investigation of the search space. The here presented Monte Carlo Tree Search algorithm (MCTS) is an adoption of the Monte Carlo Tree Search with a UCB tree selection policy (see e.g. Browne et al. [2012]) and does provide such a directed search. Since the algorithm is quite complex, it is presented in five parts that are described in the following paragraphs.

The function ***MCTS(..)*** (see Algorithm 10) represents the actual algorithm and calls the other parts. It calls the procedure *resetStatistics()* to reset all counters and temporary lists that are used to calculate the scores or to avoid generating a string value twice. The function *checkSat(..)* is a shortcut for the sequence of ***Push()***, ***Assert()***, ***Check()*** and ***Pop()*** calls to check the satisfiability of the given string value. The function *chooseMostLikelyTerminationLength()* is used to determine the termination length for the current iteration by using the information of previous runs. As input the algorithm requires the length bound of the string variable, the string variable for which the models are counted, the number of models that are used for the initialization of the search direction and the exploration factor c that is used for the calculation of the scores. An optional input is the calculation budget, e.g. a time bound. Without a calculation budget the algorithm produces the exact model count. With a calculation budget produces a lower bound of the model count.

The function ***initSearchDirection(..)*** (see Algorithm 11) initializes the search direction by generating models via the Blocking Clause Method and using them as input for the function *backpropagate(..)*. Therefore the calculation of the scores get influenced in direction of the already checked models. As input the algorithms requires the number of models that should be generated. As output the algorithm returns the number of actual generated models, which can be smaller than the requested number if the formula does not provide so much models.

The procedure ***backpropagate(..)*** (see Algorithm 12) is used to store the information of the already checked string values to influence calculation of the scores and hence the later search direction. It uses the arrays N that stores the number of visits per string value and Q that stores the number of visits per string value if the checked value was satisfiable. As input the algorithm requires the checked string value and the result of the satisfiability check.

The function ***generateNextStringValue(..)*** (see Algorithm 13) generates the next string value that will be checked with the requirement not to choose a string twice. This algorithm operates recursively based on DFS, but in order to find the next letter it does not iterate the alphabet, but calls the function *chooseNextCharacter(..)*. As input it requires the current

string value (starting with the empty string), the current recursion depth (starting with zero), the maximum recursion depth, i.e. the termination depth for this iteration, and the exploration factor c. As output it returns the generated string value or one of the following error codes: `EXHAUSTED` if there is no remaining letter that can be chosen for the given prefix or `UNSAT_PREFIX` if the given prefix is not satisfiable.

The function ***chooseNextCharacter(..)*** (see Algorithm 14) calculates the scores for the possible next letters for the given prefix and returns the letter with the highest score. There are three heuristics to calculate the score of a letter that was not chosen yet to append the given prefix: (1) *First Exploration* that first explores all possible letters before choosing one again, (2) *First Play Urgency* that first tries to exploit the already used letters and (3) *Most Likely Position* that is the *First Play Urgency* for letters that were not chosen yet for the current position and otherwise it chooses the letter that was chosen most successfully for this position. The algorithm requires as input the prefix of the searched letter, the current size of the prefix, the maximum length of the searched string and the exploration factor. As output it produces the next character or the error code `EXHAUSTED` if there is no letter left for the given prefix. If the letter was already chosen for the prefix, then the score is calculated by the following formula:

$$score := \frac{Q(prefix + letter)}{N(prefix + letter)} + c \times \sqrt{\frac{2\ ln\ N(prefix)}{N(prefix + letter)}} \tag{3.5}$$

This equation is based on the heuristic proposed by Auer et al. [2002] and is the simplest policy for calculating the *upper confidence bound* (UCB), which is popular from the fields global optimization and machine learning. The left part encourages the *exploitation* of the already collected information and the right part encourages the *exploration* of the non-visited parts of the search tree.

```
Input  : the lengthBound; the calculation budget; the string variable x for which
         the models are counted; the number of initial models mInit; the
         exploration factor c
Output : #F, the model count of F
begin
    resetStatistics();
    count := 0;

    if (checkSAT("")) then  // Check empty string.
        count++;
    end

    tmpCount := initSearchDirection(mInit);  // Initializes search direction.
    if (tmpCount ≠ mInit) then return count + tmpCount;
    else count += tmpCount;

    while (enough remaining budget) do
        terminationLength := chooseMostLikelyTerminationLength();
        if (no remaining termination length) then break;
        chosenString := generateNextStringValue("", 0, terminationLength, c);
        if (chosenString has no error value) then
            isSat := checkSAT(chosenString);
            if (isSat) then
                count++;
            end
            backpropagate(chosenString, isSat);
        else if (chosenString = EXHAUSTED) then
            prune termination length;
            continue;
        else if (chosenString = UNSAT_PREFIX) then
            prune termination length;
            continue;
        else error("Should not occur!");
    end
    return count;
end
```

Algorithm 10: Adjusted Monte Carlo Tree Search with a UCB tree selection policy (= Upper Confidence Bounds for Trees *UCT*), called as *MCTS*(lengthBound, budget, x, mInit, c)

```
Input  : the number of initial models mInit
Output : the number of models generated for initialization
begin
    count := 0;
    Push();
    sat := true;
    while (sat & count < mInit) do
        model := Model(x);
        count++;
        backpropagate(model, true);
        Assert(x ≠ model);
        sat := Check();
    end
    Pop();
    return count;
end
```

Algorithm 11: Initialization of the search direction for MCTS, called as *initSearchDirection*(mInit)

```
Input : the string value that need to be backpropagated; the satisfiability (isSat) of
        the given value
begin
    currentString := value;
    while (currentString ≠ empty) do
        N(currentString) += 1;
        if isSat then
            Q(currentString) += 1;
        end
        remove last character of currentString;
    end
    N("") += 1;
end
```

Algorithm 12: Backpropagation of checked values, called as *backpropagate*(value, sat)

```
Input  : the current string value currentStr; the current recursion depth; the
         maximum recursion depth terminationDepth; the exploration factor c
Output : generated string value or the error value EXHAUSTED and UNSAT_PREFIX
begin
    if (checkPrefixSat(currentStr) then
        if (depth < terminationDepth) then
            letter := chooseNextCharacter(currentStr, depth, terminationDepth, c);
            while (letter has no error value) do
                chosenString := generateNextStringValue(currentStr + letter, depth +
                  1, terminationDepth, c);

                if (chosenString has no error value) then return chosenString;
                else if (chosenString = EXHAUSTED) then
                    // Look for other letter.
                else if (chosenString = UNSAT_PREFIX) then
                    prune chosen letter for the termination depth;
                else error("Should not occur!");

                letter := chooseNextCharacter(currentStr, depth, terminationDepth,
                  c);
            end

            if (letter = EXHAUSTED) then
                remember that currentString exhausted for terminationDepth;
                return EXHAUSTED;
            else error("Should not occur!");

        else return currentStr;
    else return UNSAT_PREFIX;
end
```

Algorithm 13: Generation of next string value, avoids generating a string value twice, called as *generateNextStringValue*("", 0, targetLength, c)

Input : the *prefix* of the string that should be appended with the next character; the current size of the prefix (*depth*); the maximum length of the string that need to be generated (*terminationDepth*); the exploration factor *c*
Output: the selected character or the error value `EXHAUSTED`
begin
  scores := [];
  for *(letter in possible next characters)* **do** `// some characters may be pruned`
    if *letter is last letter to be added and prefix+letter was already checked* **then**
      continue; `// -> skip letter`
    else if *prefix+letter was identified as exhausted for terminationDepth* **then**
      continue; `// -> skip letter`
    else if *letter is the last but one to be added and prefix+letter is exhausted* **then**
      continue; `// -> skip letter`
    end

    if *N(prefix+letter) = 0* **then** `// choose one of the heuristics`
      `// 1. First Exploration Heuristic`
      score := `MAX_VALUE`;

      `// 2. First Play Urgency Heuristic`
      score := 0;

      `// 3. Most Likely Position Heuristic`
      if *letter was never chosen for this position* **then**
        score := 0;
      else
        score := $\frac{number\ of\ letter\ chosen\ for\ this\ position\ and\ value\ was\ satisfiable}{number\ of\ letter\ chosen\ for\ this\ position}$;
      end
    else
      score := $\frac{Q(prefix+letter)}{N(prefix+letter)} + c \times \sqrt{\frac{2\ ln\ N(prefix)}{N(prefix+letter)}}$
    end
    scores(letter) := score;
  end

  if *scores is empty* **then return** `EXHAUSTED`;
  else return $argmax(scores)$;
end

Algorithm 14: Selection of the next character in order to generate the next string value, called as *chooseNextCharacter*(prefix, depth, terminationDepth, c)

Chapter 4

Implementation

This chapter contains several decisions and problems regarding the implementation of the presented model counting techniques. The remaining part shows how the implementation was performed and describes the various algorithms that were implemented as research prototypes.

4.1 Implementation Decisions

SMT Solver

One of the most important decision for the implementation of the developed algorithms (see chapter 3.2) is the selection of the underlying SMT solver. Like described in chapter 2.3.2 there are currently two powerful SMT solvers for string constraints, namely `CVC4` [Barrett et al., 2011; Liang et al., 2014] and `Z3-STR` [Zheng et al., 2013]. In contrast to `CVC4`, `Z3-STR` does not contain the string solving theory in the actual solver core. However, the authors currently work on this, but at the moment `Z3-STR` is only available as an extension of the not up-to-date version 4.1.1 of the `Z3` SMT solver. The instructions how to build and use `Z3-STR` can be found on its GitHub page[1]. In order to be able to use the SMT solver, it has to provide a certain API that can be called by other programs. `CVC4` provides bindings for the programming languages Java and C++. This is not provided by the latest version of `Z3-STR`. Due to the needed API support and the fact that `CVC4` is at the moment the most efficient SMT solver for strings (see Aydin et al. [2015]), it is used as implementation basis for the developed algorithms. The instructions on how to build `CVC4` from source and to enable the language bindings are described on their wiki page[2]. The nightly build version *cvc4-2016-02-25* (a prerelease version of `CVC4-1.5`) was used for the presented implementation.

Programming Language

Based on the language binding support of `CVC4` the set of usable programming languages is strongly limited to Java and C++. Since the developed #SMT procedures may be deployed

[1] `https://github.com/z3str/Z3-str`

[2] `http://cvc4.cs.nyu.edu/wiki/Building_CVC4_from_source`

in the verification tool JavaPathFinder[3] and the Java community provides a lot of useful libraries e.g. for the handling of SMT constraints, this programming language was selected for the implementation of the research prototype.

Platform

The algorithms were implemented on *ubuntu 14.04 LTS* as it was necessary to build CVC4 manually to get the latest version that supports the latest string solving techniques and to generate the needed language bindings. Doing this, is much more easier on a linux machine than on a windows machine.

4.2 Problems

API Support

Discovering the API support of both solvers CVC4 and Z3-STR was a big challenge because it is not clearly documented. So manually investigation and reverse engineering was required to understand the full API and its capabilities. Python API support exists for some older version of Z3-STR, however, this was not implemented by the actual authors. The latest version of Z3 itself offers API support for a couple of languages like C, C++, .NET, Java and Python. For example Phan and Malacaria [2015] use the Java binding of Z3 to count models with respect to a set of boolean variables. But since Z3-STR does use an older version of Z3, this support is not accessible. CVC4 does support an API for Java and C++, but the libraries for the latest version are not published, so another core requirement was to manually build CVC4 and enable the generation of the language bindings.

Bugs in SMT Solvers

Besides their great functionality both SMT solvers CVC4 and Z3-STR contain several unsolved bugs/difficulties that are hard to work around. For example Z3-STR cannot solve the constraint in Listing 4.1 and answers with UNKNOWN, in contrast to CVC4 that answers with SAT (reported as issue[4]). Therefore the string operation *StartsWith* cannot be used for checking incrementally strings in a DFS algorithm based on Z3-STR. Such a problem can be solved by using a combination of another string operations, like described in Listing 4.2.

Another, more difficult problem, are appearing segmentation faults in the native code of the solvers. This happens for both of them, but especially Z3-STR has problems with the call of the ***Push()*** and ***Pop()*** operations (reported as issue[5]). Although the solver API is provided robustly by CVC4, random segmentation faults happen, resulting in programmatic

[3] http://babelfish.arc.nasa.gov/trac/jpf

[4] https://github.com/z3str/Z3-str/issues/8

[5] https://github.com/z3str/Z3-str/issues/12

restarts of the solver. This is cumbersome during the evaluation of the algorithms as the counter often needs to be restarted manually. Such problems are difficult and can be solved actually only by the developers.

Listing 4.1. SMT-LIBv2 constraint file that is not solvable with `Z3-STR` but with `CVC4`.

```
1  (declare-fun x () String)
2
3  (assert (= (str.len x) 2))
4  (assert (str.in.re x (re.* (re.range "a" "z"))))
5  (assert (str.prefixof "a" x))
```

Listing 4.2. SMT-LIBv2 constraint file showing the simulation of the string operation *Prefix/StartsWith* with the string operation *Concat* and two additional string variables.

```
1  ...
2
3  (declare-fun x () String)
4  (declare-fun y1 () String)
5  (declare-fun y2 () String)
6
7  ; x as concatenation of y1 and y2
8  (assert (= x (str++ y1 y2)))
9
10 ; prefix check as y1 assignment
11 (assert (= y1 "<prefix-value>"))
12
13 ...
```

4.3 Constraint Parsing

Using `CVC4` as binary file it provides the functionality to read SML2-LIBv2 constraint files as input. In order to be able to add programmatic constraints later via Java, e.g. for the incremental counting, it is necessary to have the constraints as internal `Expression` objects. The Java language binding for `CVC4` does not provide this functionality, so it was necessary to write a tiny parser, which reads the SML2-LIBv2 constraint files and generates internal `Expression` objects. Therefore the parser reuses the reading capability of `CVC4 Command` objects.

4.4 Algorithms

The following sections describe in which way the solution algorithms in chapter 3.2 were implemented. All of the implemented algorithms assume that the constraint contains the string variable x for that the models shall be counted. Furthermore all implemented algorithms assume that $y1$ and $y2$ are free variables in the constraint because they are used for the Concat-Trick described in the above section 4.2. As budget parameter the algorithms use a time bound specified in seconds.

Random Search

The *Random Search* algorithm was implemented following straight the pseudo code presented in chapter 3.2.1. This algorithm is used in the evaluation as baseline for the other algorithms: every search algorithm should at least perform as good as the pure random generation of string values.

Blocking Clause Method

The implementation of the *Blocking Clause* algorithm follows straight the pseudo code presented in chapter 3.2.2. The string model retrieved by `CVC4` contains quotation marks that need to be removed for the further usage.

Depth-First Search

The *standard DFS* algorithm is implemented following straight the pseudo code in chapter 3.2.3. It is is used for the scalability analysis of DFS methods for model counting of string constraints, but not for the comparison between the different solutions algorithms because it does not provide more insights than the DFS with randomized alphabet, which is described in the next section.

The *randomized DFS* from chapter 3.2.4 is separated in two algorithms. One of them is called *Depth-First Search with randomized Alphabet*, which only adds the idea from chapter 3.2.4 to the code of the standard DFS. The other is called *Randomized Depth-First Search*, which adds to the standard algorithms all randomization ideas of chapter 3.2.4: the randomized alphabet, the randomized character selection and the randomized backtracking.

The implementation of the *Depth-First Search with uniform sampled feasible Strings* algorithm follows straight the pseudo code presented in chapter 3.2.5. The function *calculateSub Set(..)* is implemented as Reservoir Sampling [Vitter, 1985]. The runtime of the sampling procedure is compared to the remaining runtime too short (see chapter 5.3 *Comparison of Solution Algorithms*) hence it makes no difference for the total runtime whether the sampling is done by a simpler sampling mechanism.

Adjusted Monte Carlo Tree Search

The implementation of the *Adjusted Monte Carlo Tree Search* algorithm follows straight the pseudo code presented in chapter 3.2.6. The *Most Likely Position* heuristic was implemented in order to calculate the score for the next letter because it performed best in the experiments.

Chapter 5

Evaluation

In order to evaluate the performed research, it is necessary to compare the developed methods with the already existing ones. As described in chapter 2 *Background* there is currently only one approach to perform model counting of string constraints, namely automata-based counting. Therefore, this chapter shows the comparison of SMT-based counting and automata-based counting. In model counting there are two main categories that need to be evaluated:

1. How fast can the models be counted? → *Performance*

2. What kind of constraints can be handled with the model counter? → *Expressiveness*

Additionally it is necessary to compare the various solution algorithms that were identified during the research. It must be evaluated which of them is the most suitable one for what kind of constraints.

All presented evaluations were executed on a virtual machine with an i7-3517U processor (1.90 GHz x2, 1.7 GB memory) running Virtual-Box *ubuntu 14.04 LTS 32-bit*, hosted with an i7-3517U processor (1.90 GHz x4, 4 GB memory) running *Windows 7 Professional Service Pack 1 64-bit*.

The remainder of this chapter is organized as follows. Section 5.1 shows the existing benchmarks that are used in model counting of string constraints. Section 5.2 shows the evaluation procedure and results for the two categories *Performance* and *Expressiveness*. Section 5.3 shows the evaluation of the various solution algorithms.

5.1 Existing Benchmarks

Like shown in Aydin et al. [2015] there are benchmarks like ASE by Kausler and Sherman [2014] and Kaluza Small/Big by Saxena et al. [2010] that contain thousands of string constraints that can be used to evaluate #SMT. The Kaluza benchmarks are taken from JavaScript programs and contain string operations like: regular expression membership, concatenation, string equality and length. ASE benchmarks are from Java programs and represent server-side code. They contain additionally constraint with the string operations: replace, indexof, contains, begins, ends and substring. These benchmarks can be used to

compare the performance with to the model counter ABC [Aydin et al., 2015] and SMC [Luu et al., 2014].

5.2 Comparison with Automata-Based Model Counters

The comparison of automata-based model counting and the presented approaches for SMT-based model counting is presented for the two main categories *Performance* and *Expressiveness*.

5.2.1 Performance

The performance of automata-based approaches is orders of magnitude faster for counting regular expression constraints than SMT-based approaches. For example Listing 5.1 shows an simple SMT-LIBv2 constraint input file with a regular expression. Table 5.1 shows the comparison of the standard DFS algorithm (see chapter 3.2.3) and the automata-based approach ABC [Aydin et al., 2015].

Listing 5.1. SMT-LIBv2 input for a scalability evaluation.

```
1  (declare-fun x () String)
2
3  (assert (str.in.re x (re.* (re.range "a" "z"))))
```

Table 5.1. Runtime results for scalability example ($|A|$=26, time bound = 3600 sec) comparing the standard DFS algorithm (see chapter 3.2.3) and the automata-based approach ABC [Aydin et al., 2015].

Length	#m=state space	Time (sec)	
		DFS	ABC
1	27	0.447	0.195
2	703	7.387	0.196
3	18,279	1403.119	0.211
4	475,255	stopped after time bound with model count: 29,392	0.262

DFS cannot keep up with automata-based counting for regular expression constraints. Furthermore the results in Table 5.1 show that the scalability of exhaustive search methods like DFS is very limited in the number of models, respectively the total solution space. Based on the example above, an exhaustive search can't handle constraints with more than approximately 18,000 models.

Nevertheless, there are special cases in which the automata-based approach takes a lot of time to calculate the generating function, especially the calculation of the determinants

is very computing expensive (see e.g. the Equation 20 in Aydin et al. [2015]). For example if the input constraint is very complex, i.e. many unions and concatenations, so that the calculation of the generating function is expensive, and the model count is zero, i.e. the constraint is not satisfiable, then #SMT is very fast compared to the automata-based approach. Listings 5.2 shows such an example and Table 5.2 shows the corresponding runtime results.

Listing 5.2. SMT-LIBv2 constraint file that leads to bad a performance of automata-based approaches.

```
1  (declare-fun x1 () String)
2  (declare-fun x2 () String)
3  (declare-fun x3 () String)
4  (declare-fun x4 () String)
5  (declare-fun x5 () String)
6  (declare-fun x6 () String)
7  (declare-fun a1 () String)
8  (declare-fun a2 () String)
9  (declare-fun x () String)
10
11 (assert (str.in.re x1 (re.+ (re.range "a" "c"))))
12 (assert (str.in.re x2 (re.+ (re.range "0" "1"))))
13 (assert (str.in.re x3 (re.+ (re.range "e" "l"))))
14 (assert (str.in.re x4 (re.+ (re.range "x" "z"))))
15 (assert (str.in.re x5 (re.+ (re.range "3" "7"))))
16 (assert (str.in.re x6 (re.+ (re.range "a" "u"))))
17 (assert (or (= a1 x1) (or (= a1 x2) (= a1 x3))))
18 (assert (or (= a2 x4) (or (= a2 x5) (= a2 x6))))
19 (assert (= x (str.++ a1 a2 x1 x2 x3 x4 x5 x6)))
```

Table 5.2. Runtime results for the input constraint of Listing 5.2, length bound = 2, comparing the standard DFS algorithm (see chapter 3.2.3) and the automata-based approach ABC [Aydin et al., 2015].

Technique	Model Count	Time (sec)
DFS	0	0.152
ABC	0	31.573

The benchmarks presented in chapter 5.1, which were used by ABC for the evaluation, are not applicable for a direct comparison to SMT-based counting because the length bounds that were used by ABC cannot be handled by #SMT, they are simply to large. They used a length bound of 50 characters with an alphabet size of 256 for both benchmarks, ASE and Kaluza Small/Big. Since the performance comparison makes not sense for these

benchmarks, the following representative selection of ASE and Kaluza benchmarks show that #SMT at least can handle the kind of constraints. The Listings 5.3 and 5.4 show extracted ASE benchmark SMT-LIBv2 constraint files of the programs *Natural CLI* and *Math Quiz Game*. The Listings 5.5 and 5.6 show Kaluza benchmark SMT-LIBv2 constraint files. The Table 5.3 show the runtime results for these benchmark files with the standard DFS algorithm (see chapter 3.2.3). (The constraints were slightly adjusted to have a variable called x that will be counted.)

Listing 5.3. Extracted ASE SMT-LIBv2 constraint file of the program *Natural CLI*.

```
1  (declare-fun x () String)
2  (declare-fun s55 () String)
3  (declare-fun i37_55_1_f () Int)
4  (assert (not (= (str.substr x 0 i37_55_1_f ) s55)))
5  (declare-fun s62 () String)
6  (declare-fun i37_62_1 () Int)
7  (declare-fun i37_62_2 () Int)
8  (assert (not (= (str.substr x i37_62_1 i37_62_2 ) s62)))
9  (declare-fun s69 () String)
10 (declare-fun i37_69_1_f () Int)
11 (assert (not (= (str.substr x 0 i37_69_1_f ) s69)))
12 (declare-fun s76 () String)
13 (declare-fun i37_76_1 () Int)
14 (declare-fun i37_76_2()  Int)
15 (assert (not (= (str.substr x i37_76_1 i37_76_2 ) s76)))
16 (declare-fun s121 () String)
17 (assert (= s121 "..." ))
18 (assert (not (= x s121 )))
19 (declare-fun s1728 () String)
20 (assert (= s1728 "..." ))
21 (assert (not (= x s1728 )))
22 (declare-fun s1730 () String)
23 (assert (= x s1730 ))
```

Listing 5.4. Extracted ASE SMT-LIBv2 constraint file of the program *Math Quiz Game*.

```
1 (declare-fun x () String)
2 (declare-fun s741 () String)
3 (assert (= s741 "y" ))
4 (assert (not (= x s741 )))
5 (declare-fun s744 () String)
6 (assert (= s744 "/restart" ))
7 (assert (not (= x s744 )))
```

```
(declare-fun s747 () String)
(assert (= s747 "n" ))
(assert (not (= x s747 )))
(declare-fun s750 () String)
(assert (= s750 "/quit" ))
(assert (not (= x s750 )))
(declare-fun s753 () String)
(assert (= s753 "/clear" ))
(assert (not (= x s753 )))
(declare-fun s756 () String)
(assert (= s756 "/setfont" ))
(assert (not (str.contains x s756 )))
(declare-fun s759 () String)
(assert (= s759 "/say" ))
(assert (not (str.contains x s759 )))
(declare-fun s762 () String)
(assert (= s762 "/setsize" ))
(assert (not (str.contains x s762 )))
(declare-fun s765 () String)
(assert (= s765 "/help" ))
(assert (not (= x s765 )))
(declare-fun s768 () String)
(assert (= s768 "/?" ))
(assert (not (= x s768 )))
```

Listing 5.5. Kaluza (sat, small) SMT-LIBv2 constraint file *1001.corecstrs.readable.smt2*.

```
(declare-fun I0_2 () Int)
(declare-fun I0_6 () Int)
(declare-fun PCTEMP_LHS_1 () Int)
(declare-fun PCTEMP_LHS_2 () Int)
(declare-fun T0_2 () String)
(declare-fun T0_6 () String)
(declare-fun T1_2 () String)
(declare-fun T1_6 () String)
(declare-fun T2_2 () String)
(declare-fun T2_6 () String)
(declare-fun T3_2 () String)
(declare-fun T3_6 () String)
(declare-fun T4_2 () String)
(declare-fun T4_6 () String)
(declare-fun T5_2 () String)
```

```
(declare-fun T5_6 () String)
(declare-fun T_2 () Bool)
(declare-fun T_3 () Int)
(declare-fun T_5 () Bool)
(declare-fun T_6 () Bool)
(declare-fun T_SELECT_1 () Bool)
(declare-fun T_SELECT_2 () Bool)
(declare-fun x () String)

(assert (= T_SELECT_1 (not (= PCTEMP_LHS_1 (- 1)))))
(assert (ite T_SELECT_1
      (and (= PCTEMP_LHS_1 (+ I0_2 0))(= x (str.++ T0_2 T1_2))
      (= I0_2 (str.len T4_2))(= 0 (str.len T0_2))(= T1_2 (str.++ T2_2 T3_2))
      (= T2_2 (str.++ T4_2 T5_2))(= T5_2 "GoogleAdServingTest=")(not
      (str.in.re T4_2 (re.++ (str.to.re "G") (str.to.re "o") (str.to.re "o")
      (str.to.re "g") (str.to.re "l") (str.to.re "e") (str.to.re "A")
      (str.to.re "d") (str.to.re "S") (str.to.re "e") (str.to.re "r")
      (str.to.re "v") (str.to.re "i") (str.to.re "n") (str.to.re "g")
      (str.to.re "T") (str.to.re "e") (str.to.re "s") (str.to.re "t")
      (str.to.re "="))))) (and (= PCTEMP_LHS_1 (- 1))(= x (str.++ T0_2 T1_2))
      (= 0 (str.len T0_2))(not (str.in.re T1_2 (re.++ (str.to.re "G")
      (str.to.re "o") (str.to.re "o") (str.to.re "g") (str.to.re "l")
      (str.to.re "e") (str.to.re "A") (str.to.re "d") (str.to.re "S")
      (str.to.re "e") (str.to.re "r") (str.to.re "v") (str.to.re "i")
      (str.to.re "n") (str.to.re "g") (str.to.re "T") (str.to.re "e")
      (str.to.re "s") (str.to.re "t") (str.to.re "="))))))))
(assert (= T_2 (not (= PCTEMP_LHS_1 (- 1)))))
(assert T_2)
(assert (= T_3 (+ PCTEMP_LHS_1 20)))
(assert (= T_SELECT_2 (not (= PCTEMP_LHS_2 (- 1)))))
(assert (ite T_SELECT_2
      (and (= PCTEMP_LHS_2 (+ I0_6 T_3))(= x (str.++ T0_6 T1_6))(= I0_6
      (str.len T4_6)) (= T_3 (str.len T0_6))(= T1_6 (str.++ T2_6 T3_6))
      (= T2_6 (str.++ T4_6 T5_6)) (= T5_6 ";")(not (str.in.re T4_6
      (str.to.re ";")))) (and (= PCTEMP_LHS_2 (- 1)) (= x (str.++ T0_6 T1_6))
      (= T_3 (str.len T0_6))(not (str.in.re T1_6 (str.to.re ";"))))))
(assert (= T_5 (= PCTEMP_LHS_2 (- 1))))
(assert (= T_6 (not T_5)))
(assert T_6)
```

Listing 5.6. Kaluza (sat, small) SMT-LIBv2 constraint file *1095.corecstrs.readable.smt2*.

```
(declare-fun I0_3 () Int)
(declare-fun I0_7 () Int)
(declare-fun PCTEMP_LHS_1 () Int)
(declare-fun PCTEMP_LHS_2 () Int)
(declare-fun T0_3 () String)
(declare-fun T0_7 () String)
(declare-fun T1_3 () String)
(declare-fun T1_7 () String)
(declare-fun T2_3 () String)
(declare-fun T2_7 () String)
(declare-fun T3_3 () String)
(declare-fun T3_7 () String)
(declare-fun T4_3 () String)
(declare-fun T4_7 () String)
(declare-fun T5_3 () String)
(declare-fun T5_7 () String)
(declare-fun T_1 () Bool)
(declare-fun T_3 () Bool)
(declare-fun T_4 () Bool)
(declare-fun T_6 () Bool)
(declare-fun T_7 () Bool)
(declare-fun T_SELECT_1 () Bool)
(declare-fun T_SELECT_2 () Bool)
(declare-fun x () String)

(assert (= T_1 (not (= "" x))))
(assert T_1)
(assert (= T_SELECT_1 (not (= PCTEMP_LHS_1 (- 1)))))
(assert (ite T_SELECT_1
      (and (= PCTEMP_LHS_1 (+ I0_3 0))(= x (str.++ T0_3 T1_3))(= I0_3
      (str.len T4_3))(= 0 (str.len T0_3))(= T1_3 (str.++ T2_3 T3_3))
      (= T2_3 (str.++ T4_3 T5_3))(= T5_3 "?")(not (str.in.re T4_3
      (str.to.re "?")))) (and (= PCTEMP_LHS_1 (- 1))(= x
      (str.++ T0_3 T1_3))(= 0 (str.len T0_3))(not (str.in.re T1_3
      (str.to.re "?"))))))
(assert (= T_3 (= PCTEMP_LHS_1 (- 1))))
(assert (= T_4 (not T_3)))
(assert T_4)
(assert (= T_SELECT_2 (not (= PCTEMP_LHS_2 (- 1)))))
(assert (ite T_SELECT_2
```

```
41          (and (= PCTEMP_LHS_2 (+ I0_7 0))(= x (str.++ T0_7 T1_7))(= I0_7
42          (str.len T4_7))(= 0 (str.len T0_7))(= T1_7 (str.++ T2_7 T3_7))
43          (= T2_7 (str.++ T4_7 T5_7))(= T5_7 "#")(not (str.in.re T4_7
44          (str.to.re "#")))) (and (= PCTEMP_LHS_2 (- 1))(= x
45          (str.++ T0_7 T1_7))(= 0 (str.len T0_7))(not (str.in.re T1_7
46          (str.to.re "#"))))))
47    (assert (= T_6 (= PCTEMP_LHS_2 (- 1))))
48    (assert (= T_7 (not T_6)))
49    (assert T_7)
```

Table 5.3. Runtime results for a representative selection of ASE and Kaluza benchmarks executed with the standard DFS algorithm (see chapter 3.2.3), $|A|$=94, time bound = 3600 sec.

Benchmark File	Length Bound	Model Count	Time (sec)
Natural CLI	1	95	25.263
Natural CLI	2	1,432	stopped after time bound
Math Quiz Game	1	93	0.703
Math Quiz Game	2	8,928	424.775
Math Quiz Game	3	27,185	stopped after time bound
1001.corecstrs.readable	21	1	33.368
1001.corecstrs.readable	22	2	stopped after time bound
1095.corecstrs.readable	2	2	2.623
1095.corecstrs.readable	3	146	stopped after time bound

5.2.2 Expressiveness

The expressiveness of a solver or model counter is limited by the constraint language it supports. Table 5.4 shows the comparison of the following solvers/model counters: `Z3-STR` by Zheng et al. [2013], `CVC4` by Barrett et al. [2011] including the extension for strings by Liang et al. [2014], `ABC` by Aydin et al. [2015], the latest version of `ABC` by Aydin et al. [2016] that is currently under development, denoted as `ABC+`, and `SMC` by Luu et al. [2014]. Since the presented #SMT approaches are implemented in `CVC4` (see chapter 4 *Implementation*), it is interesting how far the automata-based approaches can support the constraint language of `CVC4`. As described in the table, the latest version of `ABC` supports more than `SMC`, so the focus in the remained of this chapter is on `ABC`. It supports the same string operations like `CVC4`, except for the regular expression operations *Plus* and *Range*, but these can be expressed with combination of the operations *Star*, *Union* and *StringEquation*.

Table 5.4. Constraint language comparison for: Z3-STR [Zheng et al., 2013], CVC4 [Barrett et al., 2011] including the extension for strings [Liang et al., 2014], ABC [Aydin et al., 2015], the latest version of ABC [Aydin et al., 2016] that is currently under development, denoted as ABC+, and SMC [Luu et al., 2014]. (•) = SMC does not provide an operation called *IndexOf*, but the operation *strstr* that can represent the same functionality

String Operation	Z3-STR	CVC4	ABC	ABC+	SMC
Concat	•	•	•	•	•
Length	•	•	•	•	•
Substring	•	•		•	
IndexOf	•	•	•	•	(•)
StartsWith	•	•	•	•	
EndsWith	•	•	•	•	
Replace	•	•	•	•	
CharAt	•	•		•	
StringEquation	•	•	•	•	•
Contains	•	•	•	•	•
Split	•				
Regex	•	•	•	•	•
- Star	•	•	•	•	•
- Plus	•	•			
- Union	•	•	•	•	•
- Range	•	•			
- Concat	•	•	•	•	•
- In	•	•	•	•	•

Although ABC and CVC4 support the same constraint language, they do not have the same expressiveness with respect to the constraints, for which they can calculate an exact model count. Aydin et al. [2015] identifies three classes of constraints:

1. **Single-variable** constraints contain at most one string variable and therefore cannot support the string operations *Concatenation*, *Length* comparison of two variables, *CharAt*, *Substring* and *Replace*. For the remaining operations the exact count can be calculated.
2. **Pseudo-relational** constraints are multi-variable constraints, but there is at most one variable which appears in more than one clauses of a CNF formula. This variable is called projection variable and always need be on the left hand side of the constraint. This constraints can be counted exactly.
3. **Relational** constraints are multi-variable constraints with more than one variable involved in multi-variable clauses. For these constraints only an upper bound can be calculated (= over-approximation of the truth-set) by ABC because these multi-variable clauses generate a cycle in constraint evaluation.

CVC4 can calculate for all three cases an exact model count, in the boundaries of its scalability scope (see chapter 5.2.1). The following Listing 5.7 shows an SMT-LIBv2 constraint file for a relational constraint that can be counted with #SMT and Table 5.5 contains the runtime results for counting variable x.

Listing 5.7. SMT-LIBv2 input for a relational constraint

```
1  (declare-fun x () String)
2  (declare-fun y () String)
3  (declare-fun z1 () String)
4  (declare-fun z2 () String)
5
6  (assert (str.in.re x (re.+ (str.to.re "a"))))
7  (assert (= y (str.++ z1 x)))
8  (assert (= x (str.++ y z2)))
```

Table 5.5. Runtime results for the relational constraint ($|A|$=52, length bound = 5)

Algorithm	**Model Count**	**Time (sec)**
Blocking Clause	5	0.28
DFS (randomized alphabet)	5	0.283
MCTS (init=10, c=0.1)	5	0.55

Additionally, CVC4 can combine various theories in the constraints, that ABC can't support. The following Listing 5.8 shows an SMT-LIBv2 constraint file that contains a mixed-theory constraint with the theory *Strings*, *Linear Integer Arithmetics* and *Arrays*. The Table 5.6 shows the runtime results for counting variable x.

Therefore the presented model counter based on CVC4 has a greater expressiveness than automata-based model counters because it supports the calculation of the exact model count for relational constraints and it supports multi-theory constraints.

Table 5.6. Runtime results for the mixed-theory constraint ($|A|$=52, length bound = 15)

Algorithm	**Model Count**	**Time (sec)**
Blocking Clause	1	1.704
DFS (randomized alphabet)	1	59.283
MCTS (init=10, c=0.1)	1	2.18

Listing 5.8. SMT-LIBv2 input for a mixed-theory constraint constraint

```
(declare-fun x () String)
(declare-fun y () String)

(declare-fun a1 () (Array Int String))

(assert (str.in.re x (re.+ (str.to.re "a"))))
(assert (= (mod (str.len x) 3) 0))

(assert (str.in.re y (re.+ (str.to.re "c"))))
(assert (= (mod (str.len y) 5) 0))

(assert (= (store a1 0 x) a1))
(assert (= (store a1 1 y) a1))
(assert (= (str.len (select a1 0)) (str.len (select a1 1))))
```

Note: For all presented string operations in Table 5.4 ABC calculates the model count with respect to one variable. In order to support the counting of multiple independent variables they can construct various automata and combine later the counts via multiplication. The #SMT approach does calculate the model count with respect to one variable as well. The ideas on how to count formulas with respect to multiple variables are summarized in chapter 6.2 *Future Work*.

5.3 Comparison of Solution Algorithms

The presented solution algorithms (see chapter 3.2 and 4.4) need to be compared in order to find the best one or to be able to recommend one of them for a certain constraint situation. In the remainder of the chapter, the following algorithms get compared to each other:

- **Random Search**, later denoted as *Random Search* (see chapter 3.2.1).
- **Blocking Clause Algorithm**, later denoted as *Blocking Clause* (see chapter 3.2.2).
- **Depth-First Search with randomized Alphabet**, later denoted as *DFS (randomized alphabet)* (see chapter 3.2.4).
- **Depth-First Search with uniform sampled feasible Strings**, later denoted as *DFS (sampled)* (see chapter 3.2.5).
- **Randomized Depth-First Search**, later denoted das *DFS (randomized)* (see chapter 3.2.4).
- **Adjusted Monte Carlo Tree Search**, later denoted as *MCTS* (see chapter 3.2.6).

The Tables 5.7 and 5.8 show the model count characteristics of each algorithm without/with exceeding the calculation budget.

Table 5.7. Model count characteristics of every evaluated algorithm (without exceeding budget). Random Search is here only applicable with exceeding calculation budget. (•) means that the result is not fixed.

Algorithm	**Exact**	**Approximate**		**Unspecified**
		Lower Bound	**Upper Bound**	
Random Search				•
Blocking Clause	•			
DFS (randomized alphabet)	•			
DFS (sampled, ratio=1.0)	•			
DFS (sampled, ratio<1.0)	(•)	(•)	(•)	
DFS (randomized, s<1, b<1)		•		
MCTS	•			

Table 5.8. Model count characteristics of every evaluated algorithm (with exceeding budget). (•) means that the result is not fixed.

Algorithm	**Exact**	**Approximate**		**Unspecified**
		Lower Bound	**Upper Bound**	
Random Search		•		
Blocking Clause		•		
DFS (randomized alphabet)		•		
DFS (sampled, ratio=1.0)		•		
DFS (sampled, ratio<1.0)	(•)	(•)	(•)	
DFS (randomized, s<1, b<1)		•		
MCTS		•		

Three evaluation examples were identified to cover different constraint situation, in where one of the algorithms outperforms the others.

Evaluation Example 1: Huge search space, but only a few models.

The first evaluation example represents a huge search space with only a few models that are on the bottom of the search tree, so that all algorithms that need to traverse the tree until they reach the bottom, take a long time to find all models. Listing 5.9 shows the SMT-LIBv2 input for the first evaluation example.

Listing 5.9. SMT-LIBv2 input for evaluation example 1

```
(declare-fun x () String)

(assert (or
        (= x "aaaaaaaaa") (or
        (= x "bbbbbbbbb") (or
        (= x "ccccccccc") (or
        (= x "ddddddddd") (or
        (= x "eeeeeeeee")
        (= x "fffffffff") ))))))
```

Table 5.9. Runtime results for evaluation example 1 ($|A|$=52, length bound = 10, #m=6), the total runtime for DFS (sampled) is calculated by *collecting strings + building subset + checking satisfiability*. (μ is the mean, σ is the standard deviation, averaged over 10 runs)

Algorithm	**Model Count**	**Time (sec)**
Random Search (t=15 sec)	$\mu = 0,\ \sigma = 0.0$	fixed time bound
Random Search (t=30 sec)	$\mu = 0,\ \sigma = 0.0$	fixed time bound
Blocking Clause	$\mu = 6,\ \sigma = 0.0$	$\mu = 0.088,\ \sigma = 0.008$
DFS (randomized alphabet)	$\mu = 6,\ \sigma = 0.0$	$\mu = 13.463,\ \sigma = 0.493$
DFS (sampled, ratio=0.5)	$\mu = 5.8,\ \sigma = 1.887$	$\mu = 13.062,\ \sigma = 0.126$ (13.024 + 0.001 + 0.037)
DFS (sampled, ratio=0.8)	$\mu = 5.7,\ \sigma = 1.187$	$\mu = 13.077,\ \sigma = 0.332$ (13.015 + 0.001 + 0.061)
DFS (sampled, ratio=1.0)	$\mu = 6,\ \sigma = 0.0$	$\mu = 12.952,\ \sigma = 0.255$ (12.87 + 0.001 + 0.081)
DFS (randomized, s=0.01, b=0.001)	$\mu = 4.3,\ \sigma = 1.418$	$\mu = 10.251,\ \sigma = 2.423$
MCTS (init=10, c=0.1)	$\mu = 6,\ \sigma = 0.0$	$\mu = 0.139,\ \sigma = 0.032$
MCTS (init=5, c=0.1)	$\mu = 6,\ \sigma = 0.0$	$\mu = 12.072,\ \sigma = 0.224$

The runtime results for evaluation example 1 are shown in Table 5.9. For a small number of

models the Blocking Clause is highly efficient. DFS has to iterate all possible string values that needs much more time, so Blocking Clause outperforms DFS for this situation. The result for MCTS with an initialization of 10 models is as fast as Blocking Clause because there are only 6 models in total and MCTS uses Blocking Clause to generate the initial models. For a smaller initial model count it is marginal faster than DFS (randomized alphabet), which is probably caused by the small number of models. For this small number of models the DFS (sampled) is not relevant faster than the DFS (randomized alphabet). DFS (randomized) produces a good lower bound in a better time than the DFS (randomized alphabet), but without any guarantees. The Random Search counts zero models in the time bound in that the other algorithms succeeded to find all models. The search space with $\sum_{i=0}^{10} 52^i = 147,389,519,791,195,392$ string values is too large to find these six models with a complete randomized string generation in a comparable time.

Evaluation Example 2: Large search space, large model count, feasible for DFS.

The second evaluation example represents a large search space with lots of models that can be counted with a DFS algorithm. The large number of models should avoid that Blocking Clause can count them in a comparable time. Additionally this example shows the efficiency of the DFS variants. Listing 5.10 shows the SMT-LIBv2 input for the second evaluation example. The runtime results are shown in Table 5.10. DFS (randomized alphabet) can handle such constraints very efficient and is faster than MCTS. MCTS is the worst performing algorithm of the systematic search algorithms. The calculation overhead of MCTS to avoid the multiple selection of the same string value decreases the performance to find efficiently all models for this kind of scenario. DFS (sampled, ratio=0.5) gets a very good approximation of the model count, and even DFS (randomized) gets a good model count although it is not guaranteed. The Blocking Clause is for an exhaustive search of such a model counts not usable because the runtime is more than an hour. Although the search space with $\sum_{i=0}^{5} 52^i = 387,659,013$ string values is a lot of smaller than in example 1, the Random Search cannot find any of the 483 models in a comparable time.

Listing 5.10. SMT-LIB v2 input for evaluation example 2

```
(declare-fun x () String)

(assert (str.in.re x
        (re.++
                (re.opt (str.to.re "d"))
                (re.+ (re.range "a" "c"))
        )
))
```

Table 5.10. Runtime results for evaluation example 2 (|A|=52, length bound = 5, #m=483, time bound = 3600 sec), the total runtime for DFS (sampled) is calculated by *collecting strings + building subset + checking satisfiability*. (μ is the mean, σ is the standard deviation, averaged over 10 runs)

Algorithm	**Model Count**	**Time (sec)**
Random Search (t=180 sec)	$\mu = 0,\ \sigma = 0.0$	fixed time bound
Random Search (t=3600 sec)	$\mu = 0,\ \sigma = 0.0$	fixed time bound
Blocking Clause	$\mu = 379.3,\ \sigma = 27.673$	stopped after time bound
DFS (randomized alphabet)	$\mu = 483,\ \sigma = 0.0$	$\mu = 66.563,\ \sigma = 2.306$
DFS (sampled, ratio=0.5)	$\mu = 481.7,\ \sigma = 1.418$	$\mu = 64.779,\ \sigma = 1.418$ (58.574 + 0.001 + 6.204)
DFS (sampled, ratio=0.8)	$\mu = 482.3,\ \sigma = 0.458$	$\mu = 69.524,\ \sigma = 1.23$ (59.324 + 0.001 + 10.199)
DFS (sampled, ratio=1.0)	$\mu = 483,\ \sigma = 0.0$	$\mu = 70.25,\ \sigma = 2.181$ (57.663 + 0.001 + 12.586)
DFS (randomized, s=0.01, b=0.001)	$\mu = 395.8,\ \sigma = 70.555$	$\mu = 47.251,\ \sigma = 12.628$
MCTS (init=10, c=0.1)	$\mu = 483,\ \sigma = 0.0$	$\mu = 178.092,\ \sigma = 8.139$
MCTS (init=5, c=0.1)	$\mu = 483,\ \sigma = 0.0$	$\mu = 183.898,\ \sigma = 9.795$

Evaluation Example 3: Large search space, biased distributed models.

The third evaluation example represents a large search space that include biased distributed models, i.e. that there are models for every path so that a DFS needs a lot of time to iterate all paths, but there is one sub space of the search tree that includes extraordinary more models than the others. Assuming a good initialization, MCTS counts more models than DFS (randomized alphabet) for a small time bound: DFS (randomized alphabet) will take some time to find the *rich* subtree, MCTS will find it faster because it focuses its search direction and does not iterate the total search tree. Listing 5.11 shows the SMT-LIBv2 input for the third evaluation example: for all characters in the alphabet except of "z" there is only one solution per each, but if the path beginning with "z" is chosen, then there are plenty of models. The runtime results are shown in Table 5.11 and 5.12. The evaluation example 3 shows that MCTS is much more efficient than DFS for a small time bound like 30 seconds (see Table 5.12), but for a large time bound like one hour (see Table 5.11) all DFS variants outperform MCTS. For the first time the DFS (sampled) algorithm produces extraordinary results. For the large time bound it outperforms all other algorithms and for the small the time bound it outperforms the other DFS variants. The reason is that DFS (sampled) can generate much more string values in the same time and therefore it is more likely that it visits string values in the *rich* subset. MCTS has a big calculation overhead for large time bounds. For small time bounds it can use the initialization as a good search direction and focuses on similar models. Surprisingly, Blocking Clause outperforms MCTS for large time bounds and for small time bounds it is even better than DFS. This raises the question

whether it is worthwhile to investigate the SMT String Value Generation (see chapter 3.1.1) for model counting, even if its cumbersome and highly complex (see chapter 6.2 *Future Work*). Like in example 1 and 2 the Random Search cannot handle the large search space.

Listing 5.11. SMT-LIB v2 input for evaluation example 3

```
(declare-fun x () String)

(assert (or
        (str.in.re x (re.++ (str.to.re "z") (re.* (re.range "a" "c")))) (or
        (= x "aaaaaaaaa") (or (= x "bbbbbbbbb") (or (= x "ccccccccc") (or
        (= x "ddddddddd") (or (= x "eeeeeeeee") (or (= x "fffffffff") (or
        (= x "ggggggggg") (or (= x "hhhhhhhhh") (or (= x "iiiiiiiii") (or
        (= x "jjjjjjjjj") (or (= x "kkkkkkkkk") (or (= x "lllllllll") (or
        (= x "mmmmmmmmm") (or (= x "nnnnnnnnn") (or (= x "ooooooooo") (or
        (= x "ppppppppp") (or (= x "qqqqqqqqq") (or (= x "rrrrrrrrr") (or
        (= x "sssssssss") (or (= x "ttttttttt") (or (= x "uuuuuuuuu") (or
        (= x "vvvvvvvvv") (or (= x "wwwwwwwww") (or (= x "xxxxxxxxx") (or
        (= x "yyyyyyyyy") (or (= x "AAAAAAAAA") (or (= x "BBBBBBBBB") (or
        (= x "CCCCCCCCC") (or (= x "DDDDDDDDD") (or (= x "EEEEEEEEE") (or
        (= x "FFFFFFFFF") (or (= x "GGGGGGGGG") (or (= x "HHHHHHHHH") (or
        (= x "IIIIIIIII") (or (= x "JJJJJJJJJ") (or (= x "KKKKKKKKK") (or
        (= x "LLLLLLLLL") (or (= x "MMMMMMMMM") (or (= x "NNNNNNNNN") (or
        (= x "OOOOOOOOO") (or (= x "PPPPPPPPP") (or (= x "QQQQQQQQQ") (or
        (= x "RRRRRRRRR") (or (= x "SSSSSSSSS") (or (= x "TTTTTTTTT") (or
        (= x "UUUUUUUUU") (or (= x "VVVVVVVVV") (or (= x "WWWWWWWWW") (or
        (= x "XXXXXXXXX") (or (= x "YYYYYYYYY") = x "ZZZZZZZZZ")
))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Table 5.11. Runtime results for evaluation example 3 ($|A|$=52, length bound = 9, #m=9892, time bound = 3600 sec), the total runtime for DFS (sampled) is calculated by *collecting strings + building subset + checking satisfiability*. (μ is the mean, σ is the standard deviation, averaged over 10 runs)

Algorithm	**Model Count**	**Time (sec)**
Random Search (t=3600 sec)	$\mu = 0,\ \sigma = 0.0$	fixed time bound
Blocking Clause	$\mu = 525.3,\ \sigma = 12.884$	stopped after time bound
DFS (randomized alphabet)	$\mu = 287.5,\ \sigma = 287.924$	stopped after time bound
DFS (sampled, ratio=0.5)	$\mu = 755.4,\ \sigma = 19.268$	stopped after time bound
DFS (sampled, ratio=0.8)	$\mu = 750.5,\ \sigma = 15.5$	stopped after time bound
DFS (sampled, ratio=1.0)	$\mu = 759.6,\ \sigma = 17.345$	stopped after time bound
DFS (randomized, s=0.01, b=0.001)	$\mu = 398.2,\ \sigma = 324.489$	stopped after time bound
MCTS (init=10, c=0.1)	$\mu = 231.5,\ \sigma = 66.793$	stopped after time bound
MCTS (init=5, c=0.1)	$\mu = 231.5,\ \sigma = 112.399$	stopped after time bound

Table 5.12. Runtime results for evaluation example 3 ($|A|$=52, length bound = 9, #m=9892, time bound = 30 sec), the total runtime for DFS (sampled) is calculated by *collecting strings + building subset + checking satisfiability*. (μ is the mean, σ is the standard deviation, averaged over 10 runs)

Algorithm	**Model Count**	**Time (sec)**
Random Search (t=30 sec)	$\mu = 0,\ \sigma = 0.0$	fixed time bound
Blocking Clause	$\mu = 68,\ \sigma = 0.0$	stopped after time bound
DFS (randomized alphabet)	$\mu = 1.6,\ \sigma = 0.49$	stopped after time bound
DFS (sampled, ratio=0.5)	$\mu = 2.2,\ \sigma = 1.077$	stopped after time bound
DFS (sampled, ratio=0.8)	$\mu = 1.8,\ \sigma = 0.4$	stopped after time bound
DFS (sampled, ratio=1.0)	$\mu = 2,\ \sigma = 0.0$	stopped after time bound
DFS (randomized, s=0.01, b=0.001)	$\mu = 1.4,\ \sigma = 0.49$	stopped after time bound
MCTS (init=10, c=0.1)	$\mu = 37.2,\ \sigma = 3.487$	stopped after time bound
MCTS (init=5, c=0.1)	$\mu = 13,\ \sigma = 0.0$	stopped after time bound

Chapter 6

Conclusions and Future Work

This chapter contains the conclusion about the performed research and the resulting recommendations. Additionally it contains the description of the possible future work.

6.1 Conclusions

This thesis explored and evaluated #SMT procedures for model counting of string constraints. Two classes of approaches for #SMT have been investigated (see Section 3.1): *(i)* **SMT Model Generation**, adjusting the internal model generation of the SMT solver to produce all possible models *(ii)* **SMT API Usage**, using the standard smtlib operations (operation *Push()*, *Pop()*, *Assert()*, *Check()*, *Model()*) to construct an oracle, checking whether a specific string instance satisfies a constraint, and combine it with enumeration or search search procedures for candidate models to be checked *(iii)* **Constraint File Adjustment**, tweaking the smtlib input files for an efficient interaction with the SMT solver when an API is not available.

SMT Model Generation is an enumeration approach quickly overwhelmed by the complexity of numerous and increasingly complex verification tasks to be solved for the generation of the models. Exploiting the SMT solver as an oracle and combining it with appropriate enumeration or search procedures can avoid this complexity bottleneck for both exact and approximate solutions. For simplicity, this thesis is focused on the use of an SMT API; its results can be straightforwardly extended to interacting with the solver via constraint input files.

Several enumeration and search algorithms have been investigated for the generation string values as possible models for the given logical formula. An SMT solver has been used as an *oracle* to assess the satisfiability of these. There are currently two SMT solvers that can handle string constraints, namely `CVC4` [Barrett et al., 2011; Liang et al., 2014] and `Z3-STR` [Zheng et al., 2013] (see Section 2.3.2). `CVC4` has been adopted for the implementation of the algorithms because it provides the necessary API functions, and API in Java, and it demonstrated to be usually faster than `Z3-STR` in practice.

Seven algorithms have been implemented and evaluated: *Random Search*, *Blocking Clause*, *standard Depth-First Search*, *Depth-First Search (randomized alphabet)*, *Depth-First Search (sampled)*, *Depth-First Search (randomized)* and *Adjusted Monte Carlo Tree Search*.

The evaluation consists of two parts: *(i)* the comparison of #SMT with automata-based

model counting, which represents the state of the art of model counting of string constraints (e.g. ABC [Aydin et al., 2015]) and *(ii)* the comparison of the various algorithms.

As shown in Section 5.2, the *performance* of automata-based model counting, in terms of average execution time, is orders of magnitude better than #SMT. However, several classes of constraints that can be solved faster by #SMT, namely those involving a wide use of disjunction and concatenation operations (leading to an exponential growth of the automata) or complex unsatisfiable constraints (where a possibly large automata has to be built before realizing it contains no accepting paths).

Another comparison point is the *expressiveness* of the model counters, where #SMT outperforms automata-based approaches. The counter ABC, as representative of the state of the art in automata-based model counting, can handle only constraints with regular expressions, linear word equations, and the string operations presented in Table 5.4. Furthermore, for certain classes of these constraints, called *relational* constraints, automata-based model counting can only only an upper bound for the actual number of models. In contrast #SMT can calculate an exact model count for them, though possibly at a higher cost in terms of execution time. Additionally, #SMT can handle mixed-theory constraints based on the capabilities of the underlying SMT solver.

The *Random Search* algorithm (Section 3.2.1) was meant to represent a baseline for the evaluation: a search algorithm should perform better than just randomly picking string values. The evaluation data show that Random Search cannot handle the very large solution spaces of usual string constraints. This is further confirmed observing that all the other algorithms outperformed Random Search during the evaluation.

The *Blocking Clause* algorithm (Section 3.2.2) is one of the most naive approaches to perform model counting. Nevertheless it can outperform the other algorithms for constraints with a very small number of models in a huge search space.

The *Standard Depth-First Search* algorithm (Section3.2.3) is only used as implementation basis for the other DFS variants. For an exhaustive search, i.e. without a calculation budget, the standard DFS performs comparably to Depth-First Search (randomized alphabet).

The *Depth-First Search (randomized alphabet)* algorithm (Section 3.2.4) is a variant of the standard DFS where the prefix tree modeling all the possible assignments to a string variable is traversed in a randomized order. Due to its systematic and exhaustive search, Depth-First Search (randomized alphabet) represents an intuitive exact model counting solution. However, since it does not follow an informed search strategy, it may be outperformed by other search approaches able to exploit additional information about the distribution of the models in the search space, whether provided by the user or learnt automatically during the search.

The *Depth-First Search (randomized)* algorithm (Section 3.2.4) represents a full randomized version of DFS producing a lower bound approximation of the model count without guarantees. It is suitable if the time is a very crucial factor because it produces similar model counts as Depth-First Search (randomized alphabet) in a shorter time. Since it does not provide any guarantees about the approximation, it might not be suitable for the

evaluation of critical properties.

The *Depth-First Search (sampled)* algorithm (Section 3.2.5) produces an approximate model count by uniformly sampling all feasible (i.e., prefix satisfiable) strings. Since it can generate more string values than the Depth-First Search (randomized alphabet) in the same time interval, it allows to count the solutions for constraints where Depth-First Search (randomized alphabet) exceeds the time bound.

The *Adjusted Monte Carlo Tree Search* (MCTS) algorithm (Section 3.2.6) is a directed search algorithm; it is initialized with a set of models generated by the SMT solver and it automatically extends the search, possibly covering the whole state space. For MCTS, it is possible to trade off exploration for exploitation by tuning the parameters of the algorithm. In the evaluation, it only outperformed the DFS variants for non-uniform model distributions and a short time bound. However, for these constraints, the Blocking Clause performed even better than MCTS.

In conclusion there is not a *best* performing algorithm for any possible constraint. Indeed, different algorithm performs better or worse for different classes of constraints. A combination of algorithms operating on independent clauses of a complex constraint might provide a better combined solution to the problem. However, the investigation of this possibility is left as future work, and briefly sketched in the next section.

6.2 Future Work

There is no single string model counting algorithm that outperforms the others for every possible constraints. Automata-based model counting is usually the way to go when the constraint can be represented by a regular expression; however, this class of constraints is significantly restrictive. Furthermore, computationally demanding automata constructions might be a waste of time when the constraint to analyze is unsatisfiable; these cases can on the other be handled efficiently by #SMT. A preliminary analysis procedure might be devised to decide for each constraint (or independent clause of a constraint) what is the most efficient algorithm for counting its solutions. The preliminary analysis may also be used to reduce the size of the search space by pruning out regions trivially unsatisfiable.

The presented techniques for #SMT are limited on counting string constraints on one variable. Their extensions to the multi-variables case requires dealing with a significantly increased complexity, which can possibly make the discussed algorithms too inefficient in practice. There are two cases for multi-variable constraints: *(i)* constraints with independent variables and *(ii)* constraints with intertwined variables. The first case can be solved by counting separately for every independent variable and then multiplying the independent counts. The second case is more complex because it requires to consider the possible dependencies among string variables. For example, a combination via the string operation *Concat* can be dealt with by introducing a new variable that represents the concatenation; however, for an exact count, the concatenation of all the possible values for the two substring has to be considered, leading to a combinatorial explosion of the search space.

This would require improving the discussed algorithms to efficiently handle the various types of dependency.

As shown in the evaluation chapter, the Blocking Clause algorithm can outperform the other algorithms for a constraint with a small number of models in a huge search space, or for biased distributed of the models in the search space, given enough computation time. This shows that the model generation procedures of an SMT solver can be quite efficient; hence it might be worth to investigate the modification of these procedures to directly generate multiple models, bypassing the need of invoking the API methods for each new model.

Bibliography

[Auer et al. 2002] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, 2002. ISSN 1573-0565.

[Aydin et al. 2015] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification*, volume 9206 of *Lecture Notes in Computer Science*, pages 255–272. Springer International Publishing, 2015. ISBN 978-3-319-21689-8.

[Aydin et al. 2016] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting string solver. In *Graduate Student Workshop on Computing*. Computer Science, UCSB, 2016.

[Bacchus et al. 2003] F. Bacchus, S. Dalmao, and T. Pitassi. DPLL with caching: A new algorithm for #sat and bayesian inference. *Electronic Colloquium on Computational Complexity (ECCC)*, 10(003), 2003.

[Barrett et al. 2011] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, chapter CVC4, pages 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22110-1.

[Bayardo and Pehoushek 2000] R. J. Bayardo and J. D. Pehoushek. Counting models using connected components. In *In Proceedings of the AAAI National Conference*, pages 157–162, 2000.

[Birnbaum and Lozinskii 1999] E. Birnbaum and E. L. Lozinskii. The good old davis-putnam procedure helps counting models. *J. Artif. Int. Res.*, 10(1):457–477, June 1999. ISSN 1076-9757.

[Borges et al. 2014] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8.

[Borges et al. 2015] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 866–877, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8.

[Browne et al. 2012] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. ISSN 1943-068X.

[Chistikov et al. 2015] D. Chistikov, R. Dimitrova, and R. Majumdar. *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, chapter Approximate Counting in SMT and Value Estimation for Probabilistic Programs, pages 320–334. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-46681-0.

[Darbon et al. 2006] J. Darbon, R. Lassaigne, and S. Peyronnet. Approximate probabilistic model checking for programs. In *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06)*, Technical University of Cluj-Napoca, Romania, Sept. 2006.

[Darwiche 2004] A. Darwiche. New advances in compiling cnf into decomposable negation normal form. In *In ECAI*, pages 328–332, 2004.

[Filieri et al. 2013] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 622–631, 2013.

[Flajolet and Sedgewick 2009] P. Flajolet and R. Sedgewick. *Analytic combinatorics*. Cambridge University press, 2009.

[Geldenhuys et al. 2012] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 166–176, 2012.

[Godefroid et al. 2005] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340.

[Gogate and Dechter 2007] V. Gogate and R. Dechter. Approximate counting by sampling the backtrack-free search space. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1*, AAAI'07, pages 198–203. AAAI Press, 2007. ISBN 978-1-57735-323-2.

[Gomes et al. 2006] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 54. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[Gomes et al. 2007] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *IJCAI*, pages 2293–2299, 2007.

[Gomes et al. 2009] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*, pages 633–654. 2009.

[Hoeffding 1963] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[Jerrum et al. 1986] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169 – 188, 1986. ISSN 0304-3975.

[Kausler and Sherman 2014] S. Kausler and E. Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 259–270, 2014.

[King 1976] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, 1976.

[Liang et al. 2014] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 646–662, 2014.

[Loera et al. 2004] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*, 38(4):1273 – 1302, 2004. ISSN 0747-7171. Symbolic Computation in Algebra and Geometry.

[Luu et al. 2014] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 57, 2014.

[Lynce and Marques-Silva 2007] I. Lynce and J. Marques-Silva. Random backtracking in backtrack search algorithms for satisfiability. *Discrete Applied Mathematics*, 155(12):1604 – 1612, 2007. ISSN 0166-218X. SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[Nelson and Oppen 1979] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, Oct. 1979. ISSN 0164-0925.

[Nieuwenhuis et al. 2006] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, Nov. 2006. ISSN 0004-5411.

[Orso and Rothermel 2014] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2865-4.

[Parízek and Lhoták 2011] P. Parízek and O. Lhoták. *Model Checking Software: 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, chapter Randomized Backtracking in State Space Traversal, pages 75–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22306-8.

[Phan and Malacaria 2014] Q.-S. Phan and P. Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 283–292, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2800-5.

[Phan and Malacaria 2015] Q.-S. Phan and P. Malacaria. All-solution satisfiability modulo theories: Applications, algorithms and benchmarks. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 100–109, Aug 2015.

[Robert and Casella 2005] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer-Verlag New York, Inc., 2005.

[Rubinstein 1981] R. Y. Rubinstein. *Simulation and the Monte Carlo Method*. John Wiley & Sons, Inc., 1981.

[Saxena et al. 2010] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528, 2010.

[Vitter 1985] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11 (1):37–57, Mar. 1985. ISSN 0098-3500.

[Wei and Selman 2005] W. Wei and B. Selman. A new approach to model counting. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 324–339. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26276-3.

[Yu and Cova 2008] F. Yu and M. Cova. String analysis, 2008.

[Zheng et al. 2013] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature