Efficient Sampling of SAT and SMT Solutions for Testing and Verification

by

Rafael Tupynambá Dutra

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
Adjunct Assistant Professor Jonathan Bachrach
Professor Sanjit Seshia
Professor Theodore Slaman

Fall 2019
The dissertation of Rafael Tupynambá Dutra, titled Efficient Sampling of SAT and SMT Solutions for Testing and Verification, is approved:

Chair

Date

Date

Date

Date

University of California, Berkeley
Efficient Sampling of SAT and SMT Solutions for Testing and Verification

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License
by
Rafael Tupynambá Dutra
2019

To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/
Abstract

Efficient Sampling of SAT and SMT Solutions for Testing and Verification

by

Rafael Tupynambá Dutra

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

The problem of generating a large number of diverse solutions to a logical constraint has important applications in testing, verification, and synthesis for both software and hardware. The solutions generated could be used as inputs that exercise some target functionality in a program or as random stimuli to a hardware module. This sampling of solutions can be combined with techniques such as fuzz testing, symbolic execution, and constrained-random verification to uncover bugs and vulnerabilities in real programs and hardware designs. Stimulus generation, in particular, is an essential part of hardware verification, being at the core of widely applied constrained-random verification techniques. For all these applications, the generation of multiple solutions instead of a single solution can lead to better coverage and higher probability of finding bugs. However, generating such solutions efficiently, while achieving a good coverage of the constraint space, is still a challenge today. Moreover, the problem is amplified when the constraints are complex formulas involving several different theories and when the application requires more refined coverage criteria from the solutions.

This work presents three novel techniques developed to tackle the problem of efficient sampling of solutions to logical constraints. They allow the efficient generation of millions of solutions with only tens of queries to a constraint solver, being orders of magnitude faster than previous state-of-the-art samplers. First, a technique called QUICKSAMPLER, for sampling of solutions to Boolean (SAT) constraints, with the goal of achieving a close to uniform distribution. Second, a technique called SMTSAMPLER, which is designed to sample solutions to large and complex Satisfiability Modulo Theories (SMT) constraints and aims at providing a good coverage of the constraint itself. Third, a technique called GUIDEDSAMPLER, which enables coverage-guided sampling of SMT constraints, by shaping the distribution of solutions in a problem-specific basis.

The QUICKSAMPLER algorithm takes as input a Boolean constraint and uses only a small number of calls to a constraint solver in order to produce millions of samples in a few seconds or minutes. The samples satisfy the constraints with high probability (i.e., 75%),
and the invalid samples can be easily filtered out in a post-processing step. Our evaluation of QuickSampler on large real-world benchmarks shows that it can produce unique valid solutions orders of magnitude faster than other state-of-the-art sampling tools. We have also empirically verified that the distribution of solutions is close to uniform, which was our target distribution.

SMTSampler is an extension of the technique that allows efficient sampling of solutions from Satisfiability Modulo Theories (SMT) constraints. This is important, since many constraints found in practical applications are more naturally represented by SMT formulas that include theories such as arrays and bit-vectors. By working over SMT formulas directly, without encoding them into Boolean (SAT) constraints, SMTSampler is able to sample solutions more efficiently, and also achieve a better coverage of the constraint space. In our evaluation, we have also defined a new notion of coverage that better captures the diversity of SMT solutions, and have shown that SMTSampler helps improve this coverage. SMTSampler works similarly to QuickSampler, leveraging a small number of calls to a constraint solver in order to generate up to millions of stimuli. However, SMTSampler can sample random solutions from large and complex SMT formulas with bit-vectors, arrays, and uninterpreted functions. It also checks all samples for validity, only outputting valid and unique solutions to the formula. Our evaluation on hundreds of benchmarks from SMT-LIB shows that SMTSampler can handle a larger class of SMT problems, outperforming QuickSampler in the number of samples produced and the coverage of the constraint space.

GuidedSampler is an extension of SMTSampler that allows coverage-guided sampling of SMT solutions, by letting the user specify a desired set of coverage points that will shape the distribution of solutions. This is important because most current sampling techniques lack a problem-specific notion of coverage, considering only general goals such as uniform distribution, as in QuickSampler, or the coverage of the SMT formula, as in SMTSampler. However, many applications would benefit from a more specific coverage definition, for example, based on coverage points specified by the hardware designer. Our tool GuidedSampler enables this greater flexibility by using the specified coverage points to guide the sampling algorithm into generating solutions from diverse coverage classes. And even for applications where a general notion of coverage suffices, our evaluation shows that the coverage-guided sampling approach is more effective at achieving this desired coverage. GuidedSampler is thus able to efficiently generate high-quality stimuli for constrained-random verification, by sampling solutions to SMT constraints that also cover a large number of user-defined coverage classes.
To my grandfather, Geraldo Aurélio Cordeiro Tupynambá, who first taught me how to use a computer. He is an extremely intelligent and wise man, but also a good and generous person, who cares about the well-being of others. He has given me great advice, including on the choice of university and advisor for the PhD program.

One day when I was on the 7th grade, I had taken an exam for the Brazilian Mathematical Olympiad and was still struggling with one of the problems\(^1\). He read the problem and immediately came up with this beautiful construction, which was better than anything I had done in several hours. At this point I already realized how brilliant he was.

![Diagram of fractal structure](image)

Just recently he said to my grandmother that it was silly for me to be doing a PhD in Computer Science, since I “already know everything about computing”. That is far from true\(^2\), but I appreciate his confidence in me.

---

\(^1\)XXV Brazilian Mathematical Olympiad 2003, Level 1, Phase 3, Problem 3.

\(^2\)And it was much further from true when I started the PhD.
# Contents

## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td></td>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Sampling from Logical Constraints</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Sampling from SAT Constraints with QUICKSAMPLER</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Sampling from SMT Constraints with SMTSAMPLER</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Coverage-guided Sampling with GUIDEDSAMPLER</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Outline</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Background</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>SAT Constraints</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Independent Support</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>SMT Constraints</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Eager vs. Lazy SMT Solvers</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>MAX-SAT and MAX-SMT</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Weighted Sampling</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Sampling from SAT and SMT Constraints</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>QUICKSAMPLER Technique</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>SMTSAMPLER Technique</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Coverage-guided Sampling</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Formulation of Coverage-guided Sampling</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>GUIDEDSAMPLER Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>QUICKSAMPLER Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>SMTSAMPLER Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>GUIDEDSAMPLER Evaluation</td>
<td>51</td>
</tr>
</tbody>
</table>
# List of Figures

3.1 Combining two mutations over Boolean variables. .......................... 14  
3.2 Eager combination of mutations. .............................................. 19  
3.3 Combining two mutations over $v \in BV[8]$. ................................. 26  
4.1 Combining two mutations, with coverage predicates. ...................... 34  
5.1 Average time per valid sample. .............................................. 41  
5.2 Average time per valid sample, including time to check validity. ....... 42  
5.3 Unique solutions produced over same amount of time. ................... 42  
5.4 Histograms comparing the distribution of solutions. ...................... 44  
5.5 s820a_7_4 unique solutions. .............................................. 45  
5.6 enqueueSeqSK.sk_10_42 unique solutions. ................................ 46  
5.7 Speed comparison between the different approaches. ..................... 50  
5.8 Coverage comparison between the different approaches. .................. 51  
5.9 Unique classes covered: GUIDEDSAMPLER vs. baselines. ............... 56  
5.10 Unique classes over time for one benchmark (with random predicates). 58  
5.11 Uniformity over coverage classes for one benchmark (with random predicates). 58
## List of Tables

2.1 Types of SMT variables. ......................................................... 11

3.1 Example of SMT conditions to be flipped. ............................... 25

5.1 Correctness statistics for one epoch of **QuickSampler**. .......... 37

5.2 Benchmarks for evaluation of **QuickSampler**. ....................... 39

5.3 Comparison of SAT sampling algorithms. ............................... 40

5.4 Mean ratio comparisons across all benchmarks. ....................... 41

5.5 Chi-squared uniformity test. ................................................ 44

5.6 Z3 tactics for conversion into SAT. ....................................... 47

5.7 Average benchmark statistics for **SMTSampler**. ..................... 49

5.8 **SMTSampler** results over the benchmarks. .......................... 50

5.9 Average benchmark statistics for **GuidedSampler**. ................. 54

5.10 Average number of unique coverage classes (using random predicates). 55

5.11 Mean ratios of the unique coverage classes reached and uniformity metric. 57
Acknowledgments

I would like to thank my aunt and my parents for visiting me in Berkeley, my mom for the endless support, my grandmother and the whole family and friends for their warm reception every time I went home, my uncle Guilherme for first introducing me to programming in C and GNU/Linux, and my boyfriend and friends I met in Berkeley for making my life here so special.

I also want to thank the colleagues in my research group Wontae Choi, Liang Gong, Ben Mehne, Rohan Padhye, Caroline Lemieux, Kevin Läufer, Rohan Bavishi, Ed Younis and Azad for the insightful discussions and feedback. I especially thank the committee members for their feedback on my thesis proposal during the qualifying examination, as well as the help in reviewing the dissertation. Special thanks to my advisor Koushik Sen for always believing in me and supporting me through all those years, even at times when I did not believe in myself. Thanks to presidents Lula and Dilma for the substantial investments in education that brought me to where I am today, and also to Koushik, Jonathan, the ASPIRE Lab, the ADEPT Lab, and the EECS Department for generously supplementing my funding.

Research partially funded by Brazilian Science Without Borders CAPES 13245/13-9; NSF grants CCF-1423645, CCF-1409872 and CNS-1817122; DARPA CRAFT HR0011-16-C-0052; Intel Science and Technology Center for Agile Design; the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, Award Number DE-AR0000849; ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung; and ADEPT Lab industrial sponsors and affiliates Intel, Google, Siemens, SK Hynix, Apple, Futurewei, and Seagate. Any opinions, findings, conclusions, or recommendations in this work are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.
Chapter 1

Introduction

This chapter provides an introduction to the problems of sampling from logical constraints and coverage-guided sampling, with some motivation and important applications provided in Section 1.1. Sections 1.2, 1.3 and 1.4 then introduce our work in sampling from SAT constraints, SMT constraints, and coverage-guided sampling.

1.1 Sampling from Logical Constraints

Given logical constraints, the problem of generating a set of random solutions to the constraints is important both in software and hardware testing and verification. For instance, conventional symbolic execution [41, 21] and dynamic symbolic execution techniques [31, 63, 13, 12, 20, 46, 68, 3, 56, 2, 40, 61, 4, 59, 62, 32, 66, 5] generate a path constraint for each prefix of feasible execution paths in a program and use an SMT-solver to generate a solution for each such constraint. However, in practice, these techniques face scalability problems because the number of paths for any reasonable program is astronomically large. Instead of generating a single solution for the path constraint of a path prefix, one could generate multiple solutions to randomly test multiple paths having the same prefix. We call this approach constraint-based fuzzing. If multiple solutions could be generated efficiently, this would significantly speedup symbolic execution and reap the benefits of random testing [71, 72, 10, 36, 37, 54, 30], by mitigating the computational cost associated with the symbolic execution and constraint solving of those paths.

Those dynamic symbolic execution techniques, originally developed for software testing, could also be applied to hardware, both at the level of Verilog or a higher-level HDL, such as Chisel [6]. Symbolic execution executes the hardware with the inputs replaced by fresh symbolic variables and collects constraints that need to be satisfied to reach a certain execution path. Such path constraints can then be solved to generate inputs for the circuit that will exercise the target path. In the context of hardware testing, if a coverage point of interest can be specified by a path constraint, generating random stimuli that satisfy this constraint will allow a thorough exploration of the coverage point.
In the context of hardware verification, a similar idea of constrained-random verification (CRV) [52] has already been proposed to generate high-quality inputs for hardware designs. In CRV, verification engineers specify preconditions required by the hardware and other constraints based on domain-specific knowledge [73, 51]. Multiple random inputs satisfying the constraints are then generated using a stimulus generator that can sample random solutions from a constraint. Those inputs are used to drive the design under test, in an attempt to cover the design space and trigger faults. CRV is a very successful technique, being one of the most widely used verification techniques in industry.

Random sampling could also have applications in synthesis problems. For instance, in counterexample-guided inductive synthesis (CEGIS) [65], a verifier is responsible for checking the candidate expressions produced by the learner and producing counterexamples for invalid candidates. If, instead of producing a single counterexample, the verifier could sample a diverse set of counterexamples, this could help the learner in finding a valid candidate faster.

All of those applications highlight the wide importance of the problem of sampling a diverse set of solutions to logical constraints. However, despite its importance, performing this sampling efficiently is still a challenging problem today [15]. There are approaches which can provably sample according to a desired distribution [18, 34], but are expensive to run when a large number of samples is required. Other approaches use heuristics for faster sampling [69], but that can make the samples biased towards one portion of the sampling space.

We tackle this problem by developing new sampling algorithms based on a novel idea of combining mutations that can be applied to one solution in order to generate other solutions to the constraint. We have identified a common structure in real-world constraints that enables the combination of multiple such mutations in order to generate millions of new solutions. This allowed us to develop a technique QUICKSAMPLER which applies this idea to the generation of solutions to Boolean constraints, as well as a technique SMTSAMPLER that extends it to higher-level SMT formulas. Finally, our technique GUIDEDSAMPLER enables greater control of the distribution of solutions by the specification of coverage predicates. Our experimental evaluation over hundreds of benchmarks shows that our techniques can be orders of magnitude faster than previous sampling approaches.

1.2 Sampling from SAT Constraints with QuickSampler

We now discuss our work on QUICKSAMPLER to sample solutions to SAT constraints. We first chose to work over Boolean satisfiability (SAT) problems because they are conceptually simpler, and constraints from higher level domains, such as bit-vectors or other satisfiability modulo theories (SMT) problems, could be mapped into SAT with an appropriate encoding. Our goal is to efficiently generate lots of random satisfying assignments to SAT formulas, also known as SAT witnesses. We specify the desired distribution of solutions as the uniform
CHAPTER 1. INTRODUCTION

distribution, since we do not assume any problem-specific distribution is provided.

For simplicity, we do not check the generated samples for validity during the sampling phase, but we allow a post-processing step that can check each sample and filter out the invalid ones. For some application domains, such as testing, it might also be acceptable if a small fraction of the samples are not valid solutions, so the post-processing step can be optional. Our original application for the technique was in software and hardware testing, but the technique is very general, and could be applied to any scenario where SAT solutions are required. For this testing domain, the most important metric is the number of unique valid solutions generated over time. That is because each unique valid input can help cover new portions of the program and find previously unseen bugs, while repeated samples do not increase coverage.

With that in mind, we have designed QuickSampler, a new technique for efficient sampling. QuickSampler uses a small number of constraint solver calls to generate a large number of samples. QuickSampler works as follows. First, it finds a random assignment to the variables of the Boolean formula (i.e., the constraint). Such an assignment may not satisfy the formula. QuickSampler then uses a MAX-SAT solver to find a solution of the formula that is close to the random satisfying assignment. It then flips the value of each variable in the solution and again uses MAX-SAT to find another close solution of the formula. The difference between the original solution and the modified solution is called an atomic mutation. For each variable in the formula, this generates at most one atomic mutation. A small bounded number of such atomic mutations are then combined and applied to the original solution to generate a potentially new solution. We found that such combinations of small atomic mutations often results in new valid random solutions. This is because each atomic mutation identifies a small set of variables that are tightly coupled with each other, whereas the variables from two different atomic mutations are often independent. Therefore, if two such atomic mutations are combined and applied to the original solution, then the resulting solution will often satisfy the formula. The entire process is repeated several times. Since QuickSampler creates lots of solutions by simply combining atomic mutations, it avoids making frequent solver calls (which is often the bottleneck). This in turn results in quick generation of lots of random solutions.

We have implemented QuickSampler as an open-source tool. We use Z3 [24] to solve MAX-SAT queries. The samples generated by QuickSampler are not guaranteed to satisfy a given formula, but our experiments show that they are valid solutions in our benchmarks with high probability (i.e., $\geq 0.75$). QuickSampler also produces unique valid solutions orders of magnitude (i.e., $\geq 1000\times$) faster than other state-of-the-art samplers, while generating a distribution of samples which is still close to uniform. For applications which require only valid solutions, it is also possible to use our technique, by simply checking the samples for validity and filtering out the invalid ones. Our evaluation shows that QuickSampler is still faster than the other samplers, even when including this additional checking.

The main contributions of QuickSampler are:
• New technique implemented as an open source tool\(^1\) for efficient sampling from SAT formulas.

• Novel approach that is able to produce millions of solutions from only tens of solver calls.

• Evaluation against state-of-the-art techniques on a large set of complex benchmarks, showing that it is orders of magnitude faster than previous approaches, while producing a distribution which is still relatively close to uniform.

Other research groups have already leveraged QUICKSAMPLER for applications such as bug synthesis [58] and testing of configurable systems [57], showing a good potential for the applicability of the technique.

1.3 Sampling from SMT Constraints with SMTSampler

While QUICKSAMPLER works well over Boolean constraints, many constraints obtained from practical applications are more naturally expressed as Satisfiability Modulo Theories (SMT) [8] formulas. For example, constraints obtained from symbolic execution or constrained-random verification (CRV) typically use bit-vectors to represent finite-precision integer arithmetic and arrays to represent memory access. Such constraints are also getting more complex, as they are increasingly being synthesized by automated formal methods, for example by generating constraints from a high-level specification of the hardware interfaces [55].

Sampling from such SMT constraints using QUICKSAMPLER poses some challenges. It is often possible to transform such constraints into SAT problems. However, this conversion loses the high-level structure of the SMT formulas, which could be used for faster solving and to help generate more diverse solutions. To address those challenges, we have developed a new technique SMTSAMPLE that can sample solutions directly from Satisfiability Modulo Theories (SMT) [8] constraints that include high-level theories such as bit-vectors, arrays and uninterpreted functions.

The problem of finding one solution to SMT constraints is well studied, with off-the-shelf constraint solvers available [24]. SMT-LIB [7] provides a formal language and theories for specifying the constraints, as well as a large set of industrial benchmarks for evaluation. However, the problem of generating multiple diverse solutions from one SMT constraint is much less studied in literature.

One big challenge to generating random stimuli from such constraints is that they can be quite complex, involving linear and non-linear arithmetic over a large number of bit-vectors, arrays and uninterpreted functions. When solutions are sparse and non-linearly distributed,
CHAPTER 1. INTRODUCTION

traditional techniques such as MCMC samplers do not perform well, while techniques that use constraint solvers to obtain each solution become too expensive.

Another challenge is making sure the solutions are diverse and cover a large portion of the solution space. A good stimulus generator should avoid generating solutions which are only trivially different, because those are less likely to trigger new behaviors in the circuit. For example, if we have a constraint of the form \( x > 5 \lor \phi \), where \( x \) is a 32-bit integer and \( \phi \) is a complex SMT formula possibly involving \( x \) and other variables, there are billions of possible values for \( x \) which satisfy this constraint by simply satisfying the sub formula \( x > 5 \). However, producing billions of solutions which only differ in the value of \( x \) while ignoring other variables will likely not lead to new coverage and faults.

We developed a technique SMTSampler which can efficiently sample millions of solutions from an SMT formula. SMTSampler works similarly to QuickSampler, by computing simple atomic mutations that can be applied to a satisfying assignment while preserving the satisfiability of the formula. Those mutations represent minimal sets of bits that can be flipped from the SMT variables of the formula to transform one solution into another solution to the formula. However, unlike QuickSampler, SMTSampler works directly over SMT formulas including theories of bit-vectors, arrays and uninterpreted functions. We define atomic mutations for variables of each of those types, along with operations to combine them. We show that this approach can still produce valid solutions with high probability, even for large and complex SMT formulas. Another improvement over QuickSampler is that we collect as many atomic mutations as possible and then adaptively combine subsets of those mutations together, while avoiding invalid samples and enabling the generation of a large number of valid solutions to the formula. Our evaluation shows that SMTSampler can typically generate millions of solutions, using only hundreds of calls to the constraint solver.

In order to evaluate the coverage of the constraint space, we define a metric for the internal coverage of an SMT formula. The metric is defined by regarding the formula as a circuit, so that it can serve as a proxy for the coverage that would be obtained if we were testing the design under test. Our experiments show that working over the SMT formula directly generally also improves this coverage.

The main contributions of SMTSampler are:

- Develop a technique SMTSampler and implement it in an open source tool\(^2\) for efficient sampling from SMT formulas.

- Evaluate SMTSampler on a large set of complex benchmarks from SMT-LIB, comparing it against the baseline approach of converting the formula into SAT.

- Define a metric for internal coverage of SMT formulas and use it in evaluating different sampling approaches.

\(^2\)Available at https://github.com/RafaelTupynamba/SMTSampler/.
The SMTSampler technique is already being used by a hardware verification group at Stanford for the purpose of constrained-random verification.

1.4 Coverage-guided Sampling with GuidedSampler

While we noticed that working over SMT formulas directly lead to increased coverage in SMTSampler, the distribution of solutions was still far from ideal. The generated samples could still be too biased and uninteresting. We address this limitation in GuidedSampler by allowing user-specified coverage points to guide the distribution of solutions. For example, consider our example constraint of the form $x > 5 \lor \phi$, where $x$ is a 32-bit integer and $\phi$ is a complex SMT formula, possibly involving $x$ and other variables. SMTSampler might be able to cover both disjuncts, but still be more biased towards one over the other. For example, 97% of the solutions it generates might satisfy only the first disjunct $x > 5$, while 2% satisfy only the second disjunct $\phi$ and 1% satisfy both disjuncts. If there exists a bug which can only be triggered by some inputs which satisfy both disjuncts, SMTSampler might not generate enough inputs to find the bug. A better distribution of solutions would be if, for example, 1/3 of the inputs satisfied only the first disjunct, 1/3 satisfied only the second and 1/3 satisfied both.

In addition, general notions of coverage, such as uniform distribution [29, 15, 27], or internal coverage of an SMT formula [26], may not be the most appropriate for a particular problem. For example, when testing a hardware design that implements a finite-state machine, a better metric for coverage would be making sure all relevant states and transitions are reached. QuickSampler or SMTSampler, on the other hand, might be too frequently generating solutions from the most common state. Even in scenarios where a general notion of coverage is suitable, our evaluation shows that existing sampling algorithms are not always optimal in maximizing this coverage.

To address these challenges, we formulate the problem of coverage-guided sampling, where the user can specify not only the constraint that must be satisfied, but also any number of coverage predicates that will be used to guide the distribution of solutions. Each coverage predicate partitions the set of solutions in two regions, determined by whether the predicate evaluates to True or False for each solution. Taking into account all $n$ coverage predicates, the set of solutions is partitioned in up to $2^n$ different regions, which we call coverage classes. Our goal in coverage-guided sampling is to sample from each coverage class with equal weight.

We developed a technique, called GuidedSampler, which dynamically guides the search of new solutions, by using the coverage predicates to generate solutions from different coverage classes. Similarly to SMTSampler, it requires only a small number of calls to an off-the-shelf constraint solver in order to generate millions of solutions. GuidedSampler starts by finding one base solution from a random coverage class. Then, it finds some simple mutations that can be applied to this solution in order to generate another solution from a neighboring coverage class. It then combines multiple of those mutations together to generate new solutions from previously unseen coverage classes. Our evaluation shows
that GuidedSampler outperforms existing techniques in the number of coverage classes reached, both when using general coverage predicates and also problem-specific coverage predicates.

The main contributions of GuidedSampler are:

- Formally specify the problem of coverage-guided sampling.
- Develop a technique GuidedSampler and implement it in an open-source tool\(^3\) for efficient coverage-guided sampling from SMT formulas.
- Evaluate GuidedSampler against existing techniques on a large set of complex benchmarks from SMT-LIB, both using a general notion of internal coverage of SMT formulas and a problem-specific coverage notion based on random predicates.

1.5 Outline

The dissertation is organized as follows. First, Chapter 2 presents background knowledge in SAT and SMT constraints and solvers, as well as weighted sampling. Chapter 3 describes in detail the QuickSampler [27] and SMTSampler [26] algorithms we developed for efficient sampling of SAT and SMT constraints. Then, Chapter 4 describes our coverage-guided sampling algorithm GuidedSampler [25]. Chapter 5 presents the experimental evaluation of our new techniques. Finally, Chapter 6 discusses related work in sampling from logical constraints, and Chapter 7 concludes the dissertation and proposes interesting directions for future work.

\(^3\)Available at https://github.com/RafaelTupynamba/GuidedSampler/.
Chapter 2

Background

This chapter presents some background information about sampling from logical constraints which will be used in the following chapters.

2.1 SAT Constraints

A SAT constraint is defined as a Boolean formula, which is a logical formula where all the variables are of type Boolean, evaluating to either True or False (also denoted by 1 or 0). We denote by $\text{Vars}[\phi]$ the set of variables in the formula $\phi$. SAT formulas are commonly expressed using the logical operators $\land, \lor, \neg$. A literal $l_i$ is either a variable $x_i$ or its negation $\neg x_i$. A SAT formula is said to be in Conjunctive Normal Form (CNF) if it is expressed as a conjunction of disjunctions of literals, i.e., it is a formula of the form

$$\phi = C_1 \land C_2 \land \cdots \land C_n$$

where each of the $C_i$ is a disjunction of literals

$$C_i = l_1 \lor l_2 \lor \cdots \lor l_{k_i}$$

Each $C_i$ is called a clause of the formula $\phi$. One example formula in CNF format is the following:

$$(x \lor y \lor z) \land (\neg x \lor \neg y \lor z) \land (x \lor \neg y \lor \neg z) \land (\neg x \lor y \lor \neg z)$$

(2.1)

SAT formulas obtained from practical applications are commonly expressed in CNF format and some SAT solvers require CNF formulas as inputs. The benchmarks we used for evaluation of QUICKSAMPLER are provided in CNF format. However, QUICKSAMPLER can also work directly over any SAT formulas without requiring conversion into CNF.

A solution to the formula, also called satisfying assignment or SAT witness, is an assignment $\sigma$ to all the Boolean variables of the formula that makes it evaluate to True. We represent by $\sigma[v]$ the value assigned to variable $v$ in $\sigma$. For example, one solution to the example formula (2.1) would be the assignment $\sigma$ to the variables in $\text{Vars}[\phi] = \{x, y, z\}$ that
satisfies $\sigma[x] = True$, $\sigma[y] = False$ and $\sigma[z] = False$. We denote by $\phi[\sigma]$ the Boolean value resulting from evaluating $\phi$ under the assignment $\sigma$. The solutions to $\phi$ are the assignments $\sigma$ such that $\phi[\sigma]$ is True. We denote by $Sols[\phi]$ the set of solutions to the formula. This formula $\phi$ has a total of $|Sols[\phi]| = 4$ solutions, so a perfect uniform sampler for this formula would return each possible solution with probability 25%.

2.2 Independent Support

An independent support of a formula $\phi$ is a subset of the variables $S \subseteq Vars[\phi]$ which completely determines all the assignments to a formula. More specifically, given an assignment of values to the variables in the independent support $S$, there is at most one completion of this assignment to the remaining variables which satisfies the formula. For example, consider the XOR formula

$$\phi = x \oplus y = (x \land \neg y) \lor (\neg x \land y) = (x \lor y) \land (\neg x \lor \neg y)$$

We can say that the set $S = \{x\}$ is an independent support for the formula. After the value of $x$ has been assigned, there is always at most one possible solution to the formula (in this case, exactly one). We can think of all the variables that do not belong to $S$ as being dependent on the variables in $S$. In this case, for example, we need to have $y = \neg x$ for the formula to be satisfied. Note that the independent support is not unique. For this example formula, $S = \{y\}$ would also be a valid independent support.

Knowing an independent support is helpful in reducing the number of variables to which we need to assign values. That is because it is enough for the sampling algorithm to assign values only to the variables in the independent support, since all other variables can be computed from those. In many cases, an independent support arises naturally from the application. For example, when the Tseytin transformation is used to transform a combinatorial logic circuit into a Boolean formula in conjunctive normal form (CNF), auxiliary variables are introduced for all intermediate wires in the circuit. All of those auxiliary variables can be uniquely determined given the inputs to the circuit, so the inputs form an independent support. In cases when an independent support is not known for a formula, there are also methods to compute a minimal independent support for it [38].

The benchmarks we used for the evaluation of QUICKSAMPLER included information about the independent support. Many of those benchmarks had been converted into CNF format through the Tseytin transformation, so an independent support was readily available. Knowing an independent support $S$ allows QUICKSAMPLER to only assign values to the variables in $S$. For fairness, in our evaluation we compare QUICKSAMPLER to other sampler which also leverage the information about the independent support and only assign values to the variables in $S$.

Relying on an independent support could be seen as a limitation of QUICKSAMPLER, since its performance will be degraded if no independent support is provided or computed, as it will have to assign values to all the variables in the formula. However, we have shown
in our SMTSampler work that this can be overcome with simple algorithmic changes to the technique. SMTSampler successfully adapts our QuickSampler sampling algorithm to sample solutions to large and complex SMT formulas, where we need to assign values to hundreds of bit-vectors or Boolean variables (up to tens of thousands of bits in total). Our evaluation shows that SMTSampler can efficiently sample from those large and complex SMT formulas. It is also worth noting that in case we are given a very large SAT formula without any independent support information, it is possible to use a method that computes a minimal independent support for the formula [38]. Additionally, many of the large SAT formulas found in practice are obtained from a transformation that naturally produces an independent support. So it would be easy to leverage this information in sampling. However, our evaluation of SMTSampler shows that it is generally more efficient to sample over the original high-level formula directly, instead of converting it into a low-level SAT representation.

2.3 SMT Constraints

SMTSampler and GuidedSampler work over constraints coming from a subset of Satisfiability Modulo Theories (SMT) [8] formulas. More specifically, the constraints considered are in the QF_AUFBV logic of SMT, which are quantifier-free formulas over the theories of bit-vectors, bit-vector arrays, and uninterpreted functions. We define the set of variables in the formula \( \phi \) as \( \text{Vars}[\phi] = \text{Bool} \cup \text{BV} \cup \text{Array} \cup \text{UF} \), where \text{Bool}, \text{BV}, \text{Array}, and \text{UF} are the sets of variables of type Boolean, bit-vector, array, and uninterpreted function\(^1\), respectively.

The SAT formulas can then be considered as a subset of the SMT formulas which only have variables of type Boolean. All SAT formulas can be represented using only operators from combinatorial logic, such as \( \land, \lor, \neg \). The SMT formulas, on the other hand, are logical formulas with terms (variables, constant symbols and function symbols) originated not only from the SAT logic, but also from different theories. For example, the formula

\[
\text{select}(a, 011) + v > 0110
\]

contains an array variable \( a \) and a bit-vector variable \( v \), two bit-vector constants 011 and 0110, an array function \( \text{select} \), a bit-vector function \( + \), and a bit-vector predicate \( > \).

One of the theories that is commonly used in an SMT formula is the theory of fixed-size bit-vectors. Here, we denote by \( \text{BV}[n] \) the sort of bit-vectors of size \( n \). The theory of bit-vectors includes the customary arithmetical and logical operations on bit-vectors, such as additions, comparisons and bit-wise operations.

\(^1\)Technically, \text{UF} should be a set of sets, each one for each uninterpreted function type. But in this work we will just use \text{UF} to collect all the variables which are not of the previous types \text{Bool}, \text{BV}, \text{Array}, which will be collectively labeled uninterpreted functions.
Table 2.1: Types of SMT variables.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b \in \text{Bool}$</td>
<td>$\sigma[b] = \text{False}$</td>
</tr>
<tr>
<td>$v \in \text{BV}$</td>
<td>$\sigma[v] = 01100111$</td>
</tr>
</tbody>
</table>
| $a \in \text{Array}$ | $\sigma[a][x] = \begin{cases} 
0110, & \text{if } x = 001 \\
1001, & \text{if } x = 011 \\
0101, & \text{if } x = 101 \\
0010, & \text{otherwise} \end{cases}$ |
| $f \in \text{UF}$    | $\sigma[f](x,y) = \begin{cases} 
10, & \text{if } x = 0 \land y = 10 \\
01, & \text{if } x = 1 \land y = 00 \\
11, & \text{otherwise} \end{cases}$ |

Another theory common in SMT formulas is the theory of arrays, which consists of two functions $\text{select}$ and $\text{store}$ that satisfy the usual axiom

$$\text{select}(\text{store}(a,x,y),x') = \begin{cases} 
y, & \text{if } x' = x \\
\text{select}(a,x'), & \text{otherwise} \end{cases}$$

Here, $x, x' \in \text{BV}[s_x]$ are bit-vectors of a certain size $s_x$ and $y \in \text{BV}[s_y]$ is a bit-vector with a possibly different size $s_y$. Here, $a$ is an array of domain $\text{BV}[s_x]$ and range $\text{BV}[s_y]$. The function $\text{select}$ returns the value at a given index from the array, while $\text{store}$ produces an array with a new value assigned to the given index.

The theory of uninterpreted functions is a free theory, so it does not add any new axioms. Nothing is known $a \text{ priori}$ about the result of applying such a function to its arguments. For example, if $f$ is a unary uninterpreted function, we only know that we must have $f(x) = f(y)$ if $x = y$.

Table 2.1 shows example values for variables of each type, as possible values that could be assigned in a given solution $\sigma$. Again, we denote by $\sigma[v]$ the concrete assignment to $v$ under $\sigma$.

Variables in $\text{BV}$ are fixed-size bit-vectors, such as the variable $v \in \text{BV}[8]$. Arrays must have bit-vector domains and ranges, such as the array $a$, with domain $\text{BV}[3]$ and range $\text{BV}[4]$. Uninterpreted functions can have any arity. The example shows the function $f : \text{BV}[1] \times \text{BV}[2] \to \text{BV}[2]$, of arity 2. A concrete instance $\alpha = \sigma[a]$ for an array $a$ is constructed by defining its value for a finite set of indices $I(\alpha)$ and defining a default value
\(d(\alpha)\) for all other indices. In the example shown, \(I(\alpha) = \{001, 011, 101\}\) and \(d(\alpha) = 0010\). Typically, only a small number of indices will be relevant when solving a constraint, even for array domains such as \(BV[64]\), which allows \(2^{64}\) possible indices. An analogous construction is used for uninterpreted functions, where its value is defined for a finite set of argument tuples.

### 2.4 Eager vs. Lazy SMT Solvers

Two main themes in SMT solving are the eager [11] and lazy [60] approaches. Different solvers might implement one or the other, or even use a combination of eager and lazy techniques. Either approach might be more advantageous, depending on the theories and properties of the benchmarks [35].

The eager approach consists of eagerly encoding the high-level theories into a low-level Boolean representation and then using a traditional SAT solver to solve the Boolean constraint, generally by using a DPLL-style algorithm [23] for CNF formulas. For example, one possible encoding of bit-vector variables can be obtained by bit-blasting each variable into the individual bits that compose it. Other encodings are also possible and could be more efficient depending on the application. Once a solution to the SAT formula is obtained, it can then be converted back to a solution to the original SMT formula, by using the appropriate mapping between the SAT variables and SMT variables.

The lazy approach, on the other hand, works over the original SMT formula with high-level theories by using both a SAT solver and theory solvers in cooperation [60]. The SAT solver can provide assignments to the Boolean terms that would make the formula true and the theory solvers can check if those assignments are feasible by checking if there is a solution to the theory variables that generates the desired terms. Over time, each solver can provide information to the other solvers to help the search process.

Our techniques SMTSampler and GuidedSampler work directly over the SMT constraints, so they do not mandate a particular encoding into SAT. They can then be used with any desired SMT solving approach: eager, lazy or a hybrid.

### 2.5 MAX-SAT and MAX-SMT

Our techniques use MAX-SAT or MAX-SMT [53] problems, which are optimization problems over a set of hard constraints and soft constraints, as defined below.

**Definition 2.5.1.** Maximum Satisfiability Problem (MAX-SAT). Given a set of SAT formulas \(\{\phi_1, \phi_2, \ldots, \phi_m\}\), known as hard constraints, and a set of SAT formulas \(\{\psi_1, \psi_2, \ldots, \psi_n\}\), known as soft constraints, all over the same variables \(V = Vars[\phi_i] = Vars[\psi_j]\), the MAX-SAT problem

\[
\text{MAX-SAT}(\{\phi_1, \phi_2, \ldots, \phi_m\}, \{\psi_1, \psi_2, \ldots, \psi_n\})
\]
is the problem of finding an assignment $\sigma$ to the variables in $V$ which satisfies all the hard constraints $\phi_i$ and the maximum possible number of soft constraints $\psi_j$.

The MAX-SMT problem has the same definition, except that the hard constraints $\phi_i$ and soft constraints $\psi_j$ are allowed to be arbitrary SMT formulas including the supported theories. Whenever the conjunction of hard constraints $\phi_1 \land \phi_2 \land \cdots \land \phi_m$ is satisfiable, the MAX-SAT or MAX-SMT problem has a solution, and different algorithms have been proposed to efficiently solve the optimization problem and maximize the number of soft constraints satisfied [49].

Our techniques do not require any particular MAX-SAT solving algorithm. The solver we used, Z3 [24], already includes four different options of algorithms to solve MAX-SAT optimization problems [9].

## 2.6 Weighted Sampling

A natural generalization of the uniform sampling of solutions is the problem of weighted sampling, where a more general probability distribution is specified for the generated samples. In its most general form, we could specify a weight function $W : Sols[\phi] \rightarrow [0, 1]$ such that each solution $\sigma \in Sols[\phi]$ should be sampled with probability $W(\sigma)$.

In practice, more constrained weight functions can be defined that take one of some particular forms. One possibility is literal-weighted sampling [34], where a weight is assigned to each literal, so that the weight of a solution is the product of the weights of the literals that compose it. For example, for a given variable $x$, one could assign a weight of 75% to literal $x$ and 25% to literal $\neg x$. This way, solutions where $x = \text{True}$ are 3 times more favorable than solutions where $x = \text{False}$. Literal-weighted distributions cannot specify all possible weight functions, but are powerful enough for some practical applications.

For our work on GUIDEDSAMPLER, we define another distribution of solutions designed for coverage-guided sampling. Here, we assume that we are provided a set of coverage points, which are predicates $\psi_1, \psi_2, \ldots, \psi_n$ on the variables of the formula that can be used to guide the sampling algorithm. They can specify, for example, conditions that should be reached during the verification of a hardware design. Given a set of $n$ coverage points, they divide the solution space $Sols[\phi]$ into at most $2^n$ coverage classes, according to the evaluation of the predicates for each solution. For example, one class of solutions could be the one where predicate $\psi_1$ evaluates to $\text{True}$ and predicate $\psi_2$ evaluates to $\text{False}$. The goal of coverage-guided sampling, which will be presented formally in Chapter 4, is to give equal weight to the different coverage classes in sampling. This coverage-guided distribution also cannot represent all possible general weight functions, but it can be especially useful for applications which already provide coverage points. It is also possible to specify coverage points from the constraint itself, as sub-formulas of the formula $\phi$. We show that this can be used as a general way to generate diverse solutions to SMT formulas.
Chapter 3

Sampling from SAT and SMT Constraints

In this chapter we introduce our techniques QuickSampler [27] and SMTSampler [26] for efficient sampling of solutions to SAT and SMT constraints.

3.1 QuickSampler Technique

Given a Boolean formula $\phi$, the goal of QuickSampler is to generate unique solutions of $\phi$ efficiently. Another goal of QuickSampler is to make sure that solutions of $\phi$ are sampled almost uniformly at random. The key idea behind QuickSampler is to make a small number of solver calls to generate a large number of potentially unique solutions of $\phi$. The core algorithm behind QuickSampler works as follows. QuickSampler assumes that we are given an initial random solution $\sigma$ (i.e., a satisfying assignment to $\phi$). In our algorithm, this base solution $\sigma$ is computed as the closest solution to a random assignment

$$\sigma : \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1$$

$$\delta_a : \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0$$

$$\sigma_a = \sigma \oplus \delta_a : \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1$$

$$\delta_b : \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0$$

$$\sigma_b = \sigma \oplus \delta_b : \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$$

$$(\delta_a \lor \delta_b) : \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0$$

$$\tilde{\sigma} = \sigma \oplus (\delta_a \lor \delta_b) : \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1$$

Figure 3.1: Combining two mutations over Boolean variables.
\( \sigma' \), as will be described later. Solution \( \sigma \) can be represented as a vector of 1s and 0s, where each location corresponds to a Boolean variable in \( \phi \) and the value at that location in the vector denotes the value assigned to this variable in the solution \( \sigma \). Let \( \text{Bool} \) be the set of all Boolean variables in \( \phi \). For example, in Figure 3.1 we show a possible vector \( \sigma \), in a case where the number of variables is \( |\text{Bool}| = 12 \).

For each variable \( v \in \text{Bool} \), \text{QuickSampler} finds a solution \( \sigma_v \) such that \( \sigma_v \) and \( \sigma \) are minimally different and \( \sigma_v[v] \neq \sigma[v] \), where \( \sigma[v] \) is the value of the variable \( v \) in the solution \( \sigma \). Note that such a solution may not exist for all variables in \( \text{Bool} \). The diff between \( \sigma \) and \( \sigma_v \), which we will denote using \( \delta_v \) and which is the XOR of \( \sigma \) and \( \sigma_v \), is called an atomic mutation of \( \sigma \). That is \( \delta_v = \sigma_v \oplus \sigma \). In the example from Figure 3.1, if the first variable of the formula is \( a \), we might find a new solution \( \sigma_a \) which has the first bit flipped (corresponding to variable \( a \)) and additionally other two bits flipped. The corresponding atomic mutation \( \delta_a \) is also shown in Figure 3.1. Similarly, if the second variable of the formula is \( b \), we might find a new solution \( \sigma_b \) as shown in Figure 3.1, which has the second bit (corresponding to variable \( b \)) flipped, but also other 3 bits flipped. The corresponding atomic mutation \( \delta_b \) is again shown in Figure 3.1.

By definition, the atomic mutation \( \delta_v \) always ensures that at least \( \delta_v[v] \) is one, i.e., \( \sigma \) and \( \sigma_v \) at least differ in the value of the variable \( v \) and difference in the values of the remaining variables is minimal. We will later explain how a MAX-SAT query to a SAT solver can be used to find \( \sigma_v \) given \( \phi, \sigma \), and \( v \). Given \( \sigma \), \text{QuickSampler} first computes the set of all atomic mutations by going over all the variables \( v \in \text{Bool} \). Let us denote the set of all such atomic mutations by \( \Delta^1_{\sigma} \). Note that given \( \sigma \) and \( \delta_v \), we can compute \( \sigma_v \) as \( \delta_v \oplus \sigma \).

After computing \( \Delta^1_{\sigma} \), \text{QuickSampler} computes sets of composite mutations \( \Delta^k_{\sigma} \) for \( k > 1 \), where \( \Delta^k_{\sigma} \) contains the bit-wise OR of all sets of \( k \) distinct mutations in \( \Delta^1_{\sigma} \). For example, if \( \delta_a \) and \( \delta_b \) are two mutations in \( \Delta^1_{\sigma} \) such that \( a \neq b \), then \( \delta_a \lor \delta_b \) is a mutation present in \( \Delta^2_{\sigma} \). (Since each of \( \delta_a \) and \( \delta_b \) are bit-vectors, \( \delta_a \lor \delta_b \) is computed by taking bit-wise OR of the two bit-vectors.) For example, after computing the atomic mutations \( \delta_a, \delta_b \in \Delta^1_{\sigma} \) from Figure 3.1, the combined mutation \( \delta_a \lor \delta_b \) is added to \( \Delta^2_{\sigma} \). If we apply the combined mutation to \( \sigma \), by computing \( \sigma \oplus (\delta_a \lor \delta_b) \) we obtain a new assignment \( \tilde{\sigma} \), as in Figure 3.1. Note that \( \tilde{\sigma} \) differs from \( \sigma \) on all the bits set in either of the two atomic mutations \( \delta_a \) and \( \delta_b \).

This new assignment \( \tilde{\sigma} \) is not guaranteed to be a valid solution, but we have found that it has a high probability of being valid in real benchmarks\(^1\). This is because the differences \( \delta_a \) and \( \delta_b \) consist of a minimal set of bits which can be flipped while still preserving the satisfiability of the formula. So the bits in \( \delta_a \) are likely to be closely related to each other by some clauses in the formula. Such clauses remain true when those bits are flipped all together, but are not true when a smaller number of bits is flipped. It is likely that those clauses would still be satisfied in \( \sigma \oplus (\delta_a \lor \delta_b) \), where we flip all the bits from \( \delta_a \) in addition to the bits from \( \delta_b \).

In general, each mutation \( \delta \) present in a \( \Delta^k_{\sigma} \) denotes a composite mutation and can be

---

\(^1\)Our heuristic to generate samples exploits the clause structure found in real-world benchmarks. We expect it to perform poorly if applied to a randomly-generated SAT formula.
XORed with \( \sigma \) to get an assignment \( \tilde{\sigma} \) to the variables in \( \phi \). Such an assignment may or may not be a solution of \( \phi \). Surprisingly, in our experiments we found that for small values of \( k \) (i.e., \( k \leq 6 \)), more than 73% of such assignments obtained by XORing are actual solutions of \( \phi \). Let us denote the assignments obtained by applying all the mutations present in \( \Delta^k_\sigma \) to \( \sigma \) by \( \Sigma^k_\sigma \), i.e.,

\[
\Sigma^k_\sigma = \{ \delta \oplus \sigma \mid \delta \in \Delta^k_\sigma \}
\]

We let \( \Sigma_\sigma = \bigcup_{1 \leq k \leq 6} \Sigma^k_\sigma \). We found experimentally that over all benchmarks, 75% of the assignments in \( \Sigma_\sigma \) are solutions of \( \phi \).

We now make a few interesting and important observations about the set of assignments \( \Sigma_\sigma \). **QuickSampler** needs to make solver calls only to compute \( \Delta^1_\sigma \). Moreover, it is not always necessary to make a solver call while computing the elements of \( \Delta^1_\sigma \)—if **QuickSampler** flips the bit corresponding to the variable \( v \) in \( \sigma \) and discovers that the resulting bit-vector is a satisfying assignment to \( \phi \), then **QuickSampler** can skip the solver call for \( \delta_v \). For the computation of all other \( \Sigma^k_\sigma \) with \( k > 1 \), **QuickSampler** needs no solver calls because each element in \( \Sigma^k_\sigma \) is obtained by applying at most \( k \) bit-wise Boolean operations. An assignment in \( \Sigma^k_\sigma \) may or may not be a solution to the formula, however checking its validity is fast and takes linear time in the size of \( \phi \). In summary, **QuickSampler** can potentially make solver calls for the computation of \( \Sigma^1_\sigma \), but it makes no solver calls to compute the remaining sets \( \Sigma^k_\sigma \). Another observation is that size of \( \Sigma^k_\sigma \) could grow exponentially with \( k \). This allows **QuickSampler** to rapidly generate lots of unique solutions of \( \phi \) by making very few solver calls. From only \( |\text{Bool}| \) calls to the constraint solver, we can produce a large number of assignments in \( \Sigma_\sigma \), and a significant fraction (i.e., 75%) of those have been empirically found to be solutions of \( \phi \). This forms the crux of **QuickSampler**’s core algorithm for sampling.

Given a random solution \( \sigma \), we described how **QuickSampler** generates lots of solutions that are small mutations of \( \sigma \). We next describe how we generate a random solution \( \sigma \). **QuickSampler** first chooses a random assignment \( \sigma' \) by picking the values of variables in \( \text{Bool} \) uniformly at random. Then it uses a MAX-SAT query to find a closest solution \( \sigma \) to the random assignment \( \sigma' \). We picked this strategy to make sampling of solutions more uniform. Overall, **QuickSampler** execution is divided into epochs. In each epoch, **QuickSampler** generates a random solution \( \sigma \) using the method described above. Then it computes \( \Sigma_\sigma \) and outputs the elements of \( \Sigma_\sigma \). **QuickSampler** repeats this process in a loop until it has run out of time budget or it has finished generating a user-specified number of solutions.

Now we describe how MAX-SAT queries can be used to obtain the random solution \( \sigma \) and also to obtain the solutions \( \sigma_v \) for each variable \( v \). As presented in Section 2.5, the maximum satisfiability problem, or MAX-SAT, is defined as follows: given a set of hard constraints and a set of soft constraints, find a solution which satisfies all the hard constraints and additionally satisfies the maximum possible number of soft constraints. In order to compute the random solution \( \sigma \), we just need to specify one hard constraint that the formula \( \phi \) must be satisfied and \( |\text{Bool}| \) soft constraints indicating that the values of each variable \( v \) should preferably be equal to their respective values in the random assignment \( \sigma' \), i.e., \( \forall u \in \text{Bool} : \sigma[u] = \sigma'[u] \). In order to compute each solution \( \sigma_v \), we specify two
hard constraints and \(|\text{Bool}| - 1\) soft constraints. The hard constraints are that the formula \(\phi\) must be satisfied and that the value of variable \(v\) must be flipped, i.e., \(\sigma[v] \neq \sigma'[v]\). The soft constraints are that the values of other variables should preferably remain the same, or \(\forall u \in \text{Bool}\setminus \{v\} : \sigma[u] = \sigma'[u]\).

**QuickSampler Algorithm**

Algorithm 1 shows the complete QuickSampler algorithm in pseudocode. In the main QuickSampler function, for each epoch, we generate the random assignment \(\sigma'\) and find the closest solution \(\sigma\). Then, function sample is responsible for obtaining new samples from the base solution \(\sigma\). It first outputs solution \(\sigma\). Then, in a loop, it uses MAX-SAT solver calls to try to flip each of the variables \(v\). Whenever it finds a previously unseen neighboring solution \(\sigma_b\), QuickSampler outputs it and tries to combine its corresponding atomic mutation \(\delta_b\) with the previously seen atomic mutations \(\delta_a\), outputting a large number of newly generated samples. The combine function implements the procedure described in Figure 3.1 to combine the atomic mutations corresponding to \(\sigma_a\) and \(\sigma_b\), generating a new sample \(\tilde{\sigma}\). If desirable, the output function can verify if the samples are valid or not and filter out the invalid ones before outputting them. It could also check for uniqueness of samples among all epochs, avoiding any repetition of samples.

We now describe Algorithm 1 in more detail. Our sampling algorithm is divided into epochs, with each epoch being associated with a base solution \(\sigma\). The main sampling function QuickSampler(\(\phi, S\)) works by repeatedly calling sample(\(\sigma, \phi, S, \text{unsatVars}\)) to perform one epoch of sampling. For each epoch, we choose a base solution \(\sigma\) as follows. First, we choose a random assignment \(\sigma'\) of variables in \(S\), covering the whole space uniformly. Then, function getConditions(\(\phi, \sigma', S\)) is used to collect the set \(C_{\sigma'} = \{v = \sigma'[v] \mid v \in S\}\) of conditions of the form \(v = \sigma'[v]\) which are true for assignment \(\sigma'\). Finally, we use a MAX-SAT query to find the closest solution \(\sigma\) to this random assignment. This strategy is chosen to make the sampling more uniform, allowing the whole space of assignments to be explored.

Function sample(\(\sigma, \phi, S, \text{unsatVars}\)) implements one epoch of the sampling algorithm as follows. First, we output the base solution \(\sigma\). We again use getConditions(\(\phi, \sigma, S\)) to collect the set \(C_{\sigma} = \{v = \sigma[v] \mid v \in S\}\) of conditions which are true for assignment \(\sigma\). Then, for each variable \(v\) in the independent support \(S\), we use a MAX-SAT query to find a new solution \(\sigma_b\) which is as close as possible to \(\sigma\), but has the value of \(v\) flipped. Those neighboring solutions correspond to atomic mutations which can be combined to generate new samples. We use the set neighbors to collect those neighboring solutions. Through the whole epoch, we keep a map mutations (implemented as a hash table) which collects all samples produced so far, mapping them to the number of atomic mutations which were combined to produce them.

Whenever a new neighboring solution \(\sigma_b\) is found, we output it and create a map newMutations of new samples which were discovered using \(\sigma_b\). We add \(\sigma_b\) to newMutations as a new sample of level 1 and, for every sample \(\sigma_a\) of level \(l < 6\) in mutations, we combine \(\sigma_a\) with \(\sigma_b\), generating a new sample \(\tilde{\sigma}\). If \(\tilde{\sigma}\) is a previously unknown sample (not present
Algorithm 1 QuickSampler Algorithm.

1: function QuickSampler(φ, S)
2:   unsatVars ← {}  
3: while not DONE do
4:   σ' ← GENERATERANDOMASSIGNMENT(φ, S)
5:   Cσ' ← getConditions(φ, σ', S)
6:   σ ← MAX-SAT({φ}, Cσ')
7:   if not σ then break
8:   sample(σ, φ, S, unsatVars)
9: end

10: function sample(σ, φ, S, unsatVars)
11:   output({σ})
12:   Cσ ← getConditions(φ, σ, S)
13:   neighbors ← {}
14:   mutations ← new map(assignment → int)
15:   for v in S where v ∉ unsatVars do
16:     σ_b ← MAX-SAT({φ, v ≠ σ[v]}, Cσ\{v = σ[v]})
17:     if σ_b then
18:       if σ_b ∉ neighbors then
19:         neighbors ← neighbors ∪ {σ_b}
20:         newMutations ← new map(assignment → int)
21:         newMutations[σ_b] = 1
22:         output({σ_b})
23:       for σ_a in mutations where mutations[σ_a] < 6 do
24:         ṡ ← COMBINE(σ, σ_a, σ_b, S)
25:         if ṡ ∉ mutations.keys() ∪ newMutations.keys() then
26:           newMutations[ṡ] = mutations[σ_a] + 1
27:           output({ṡ})
28:         for ṡ in newMutations.keys() do
29:           mutations[ṡ] ← newMutations[ṡ]
30:       else
31:         unsatVars ← unsatVars ∪ {v}
32:     end
33:   end
34: end
35: function COMBINE(σ, σ_a, σ_b, S)
36:   ṡ ← new assignment()
37:   for v in S do
38:     δ_a ← σ[v] ⊕ σ_a[v]
39:     δ_b ← σ[v] ⊕ σ_b[v]
40:     ṡ[v] ← σ[v] ⊕ (δ_a ∨ δ_b)
41:   return ṡ
in \textit{mutations} or \textit{newMutations}), we output it and add it to \textit{newMutations} as a new sample of level \(l + 1\). Finally, the samples from \textit{newMutations} are merged into \textit{mutations}. This algorithm allows the generation of samples of level at most 6. We do not further combine mutations of level 6 with new atomic mutations because the chances that the resulting samples will satisfy the formula might be too low.

We next describe some important optimizations incorporated into the QuickSampler algorithm, namely the eager generation of samples, the use of the independent support of the formula, and the removal of unsatisfiable variables.

\textbf{Eager Generation of Samples}

In QuickSampler, we have decided to compute the combined mutations as soon as a new atomic mutation is known because this way our algorithm can output new samples earlier, without waiting for the completion of several MAX-SAT calls. On the largest benchmarks, each solver call can be slow to complete, so it may be better to output all possible samples we can from the solver calls which are already completed. We also found it essential to avoid duplicates within one epoch, by keeping track of currently known \textit{mutations}. Otherwise, we would output too many repeated samples and perform unnecessary work computing them.

Figure 3.2 displays this eager generation. Each circle represents one mutation and inside it we indicate the number of atomic mutations used to generate it. When solution \(\sigma_a\) is returned by the solver, we learn one atomic mutation \(\delta_a\), represented by the first circle in the figure. Then, as soon as solution \(\sigma_b\) becomes available, we learn the atomic mutation \(\delta_b\) and also combine it with \(\delta_a\) to generate a mutation in \(\Delta^2_a\). Then, as soon as solution \(\sigma_c\)
becomes available, we learn the mutation $\delta_c$ and combine it with the three previously known mutations in order to generate three new mutations. This is implemented as the nested loop in line 23 of Algorithm 1. Whenever a new neighboring solution $\sigma_b$ is found, it is immediately combined with the previously known samples $\sigma_a$ from the current epoch.

In conjunction with this eager generation of mutations, we also eliminate duplicate mutations in the current sampling epoch. This is done by checking in line 18 of Algorithm 1 if the new neighboring solution $\sigma_b$ has already been found in the current epoch and checking in line 25 if the newly generated sample $\tilde{\sigma}$ was already seen in the current epoch.

**Independent Support**

Similarly to UniGen2 [15], we can restrict our sampler to only operate over the variables in an independent support $S$ of the formula, instead of generating assignments to all the variables in $\text{Bool}$. As described in Section 2.2, the independent support is a subset of variables which completely determines all the assignments to a formula. More specifically, given an assignment of values to the variables in the independent support $S$, there is at most one completion of this assignment to the remaining variables which satisfies the formula. So we can think of all other variables being dependent on the variables in the independent support. Knowing an independent support is helpful in reducing the number of variables for which we need to assign values.

In Algorithm 1, we only need to loop over the variables of the independent support $S$ in line 15. Also when combining mutations in function $\text{combine}$, we only need to assign values to the variables in $S$.

**Unsatisfiable Variables**

If one MAX-SAT query $\text{MAX-SAT}(\{\phi, \ v \neq \sigma[v]\}, C_\sigma)$ for variable $v$ returns no solutions, we learn that $v$ can only have one value in this formula. When this happens in the first epoch, we record the variable $v$ in a set $\text{unsatVars}$ of unsatisfiable variables. Then, we do not try to flip the value of $v$ again in other epochs. We found that, over all benchmarks, on average 6% of the variables from the independent support were added to the set $\text{unsatVars}$. This means that, after the first epoch, all subsequent epochs can work over a reduced sampling set and avoid unnecessary solver calls.

### 3.2 SMTSampler Technique

We now describe SMTSampler, our extension of the QuickSampler technique to work over SMT formulas. One approach that could be used to sample solutions from an SMT formula is to first convert it into a SAT constraint. Then, an existing SAT sampler such as QuickSampler could be used to sample solutions from the SAT formula. Such solutions can then be converted back into solutions to the original SMT constraint. This is an eager
strategy of SMT encoding into SAT. However, our experiments show that working over
the SMT constraint directly enables greater efficiency in sampling and better coverage of
the constraint space for most benchmarks. That is why we developed SMTSAMPLER, a
technique to sample solutions from the SMT formula directly, without requiring a conversion
into SAT. This enables the constraint solver to use any preferred encoding and solving
strategy, without mandating one particular SAT encoding. This way, we can take advantage
of lazy SMT approaches and more efficient encoding to capture bit-vector equivalences, bit-
vector operations and array operations.

Just like QuickSampler, SMTSAMPLER uses a small number of calls to an off-the-
shelf constraint solver in order to generate a large number of solutions. The core idea is
still learning interesting ways that the solutions of a formula can be modified minimally to
generate new solutions (atomic mutations). We define a combination function which can be
used to merge the effects of several distinct atomic mutations over SMT variables of type
Boolean, bit-vector, array and uninterpreted function. It generates a compound mutation
and applies it to the original solution, producing a possibly new solution. The combination
function can be leveraged to generate millions of samples from just a few hundreds of atomic
mutations. The samples generated by the combination function are assignments which may
or may not satisfy the formula. However, our experiments show that they have a high
probability of satisfying the formula, even on large and complex industrial benchmarks.
Moreover, unlike QuickSampler, SMTSAMPLER checks each generated sample for validity
and only outputs valid solutions.

SMTSAMPLER works similarly to QuickSampler, except for the following differences.
First and foremost, since we want to sample directly from SMT formulas, the combination
function that combines mutations to generate new samples had to be extended to work over
variables of type bit-vector, array and uninterpreted function. Second, SMTSAMPLER only
outputs valid solutions and it replaced the eager generation of samples in QuickSampler by
a new strategy of adaptive generation of samples based on accuracy. This change was required
because the simpler approach of outputting samples without checking them for validity was
producing valid samples with very low probability for large and complex SMT formulas.
By checking all samples for validity, only using valid samples in the combination function
to generate new samples, and finishing a sampling epoch early when accuracy becomes too
low, SMTSAMPLER was able to match the high accuracy of QuickSampler (75%) for its
generated samples, even on large and complex SMT benchmarks. Finally, one last change
that was required in SMTSAMPLER was the use of timeouts for scalability reasons. We
established a limit on the amount of time that can be spent computing neighboring solutions
in one single epoch. That is because some benchmarks have a very large number of variables
and it is not feasible to try to flip each bit in the base solution. We also applied a timeout
of 5 seconds for each MAX-SMT solver call because some MAX-SMT constraints were too
expensive to solve.

Next we present the full details of the algorithm. First, we describe the main sampling
procedure of SMTSAMPLER. Next, we explain how one base solution is chosen for each
epoch, and how we discover a set of neighboring solutions to the base solution. Finally, we
describe how those solutions are used to generate new samples.

**SMTSampler Algorithm**

**Algorithm 2** SMTSampler algorithm.

```
1: function SMTSampler(φ)
2:    while not DONE do
3:        σ' ← generateRandomAssignment(φ)
4:        Cσ' ← getConditions(φ, σ')
5:        σ ← MAX-SMT({φ}, Cσ')
6:        output({σ})
7:    Sigma ← computeNeighboringSolutions(φ, σ)
8:    output(Sigma)
9:    α ← 1, k ← 1, Sigma ← Sigma
10: while α ≥ α_{min} ∧ k < 6 do
11:    (Sigma_{k+1}, α, Sigma) ← combineMutations(Sigma_k, Sigma_1, Sigma, φ)
12:    output(Sigma_{k+1})
13:    k ← k + 1
14:
15: function computeNeighboringSolutions(φ, σ)
16:    Cσ ← getConditions(φ, σ)
17:    Sigma ← {}  
18:    for c in Cσ do
19:        σ̃ ← MAX-SMT({φ, ¬c}, Cσ\{c})
20:        if σ̃ then
21:            Sigma ← Sigma ∪ {σ̃}
22:    return Sigma
23:
24: function combineMutations(Sigma_k, Sigma_1, Sigma, φ)
25:    valid ← 0, checks ← 0
26:    for (σ_a, σ_b) in Sigma_k × Sigma_1 do
27:        σ̃ ← Ψ_σ(σ_a, σ_b)
28:        if σ̃ ∉ Sigma then
29:            Sigma ← Sigma ∪ {σ̃}
30:            checks ← checks + 1
31:        if φ[σ̃] then
32:            Sigma_k+1 ← Sigma_k+1 ∪ {σ̃}
33:            valid ← valid + 1
34:    return (Sigma_k+1, valid/checks, Sigma)
```
Algorithm 2 presents the main SMTSampler procedure, which takes as input an SMT formula $\phi$. Just like QuickSampler, the SMTSampler algorithm works over several epochs. In each epoch, we first sample one initial base solution $\sigma$. This is done by generating a random assignment $\sigma'$ to the variables of the formula in line 3 and then using MAX-SMT to obtain the solution $\sigma$ which is closest to $\sigma'$ in line 5. Then, in line 7, we use the function computeNeighboringSolutions to compute a set $\Sigma^1_\sigma$ of neighboring solutions for $\sigma$. This function will also be described later.

As in QuickSampler, the core idea in SMTSampler is the combination of mutations to generate new samples. We define a combination function $\Psi : X \times X \times X \rightarrow X$, where $X$ is the space of all possible assignments to the variables $\text{Vars}[\phi]$ in the formula. We denote by $\Psi_\sigma(\sigma_a, \sigma_b)$ the application of the combination function to the base solution $\sigma$ and two other solutions $\sigma_a$ and $\sigma_b$. Intuitively, the combination function $\Psi$ computes the mutations which can be applied to $\sigma$ to generate $\sigma_a$ and $\sigma_b$, then merges those two mutations together to produce a new assignment. In QuickSampler, the combination function would be defined as

$$\Psi_\sigma(\sigma_a, \sigma_b)[v] = \sigma[v] \oplus ((\sigma[v] \oplus \sigma_a[v]) \lor (\sigma[v] \oplus \sigma_b[v]))$$

For SMTSampler, we provide a natural extension of this definition that can handle bitvectors, arrays and uninterpreted functions. The assignment returned by $\Psi$ is not guaranteed to satisfy the formula, but in practice it is a solution with high probability. This is because the atomic mutations capture the minimal changes that preserve the satisfiability of the formula, and we designed $\Psi$ to combine those changes in an additive way. The full definition of $\Psi$ will be presented later.

We next describe how function combineMutations uses $\Psi$ to generate new samples. We denote by $\Sigma^1_\sigma$ the set of neighboring solutions to $\sigma$ obtained from computeNeighboringSolutions. Starting from $\Sigma^1_\sigma$, our goal is to compute sets $\Sigma^k_\sigma$ which will contain solutions generated by combining $k$ atomic mutations, for $1 \leq k \leq 6$. Throughout the current epoch, we maintain a set $\Sigma_\sigma$ of samples which were computed so far, both valid and invalid. Initially, $\Sigma_\sigma = \Sigma^1_\sigma$. There is no need to check the elements of $\Sigma^1_\sigma$, since we know they are valid samples, as they were obtained directly from the constraint solver.

Now assume that we already constructed a set $\Sigma^k_\sigma$. We can inductively build the set $\Sigma^{k+1}_\sigma$ as follows. For each pair of samples $\sigma_a \in \Sigma^k_\sigma$ and $\sigma_b \in \Sigma^1_\sigma$, we apply the combination function $\Psi$ to generate a new sample $\tilde{\sigma} = \Psi_\sigma(\sigma_a, \sigma_b)$. If $\tilde{\sigma}$ is an element of $\Sigma_\sigma$, it has already been checked and is discarded. Otherwise, we add it to $\Sigma_\sigma$ and check if it is a solution to the formula. This checking is relatively fast, as it only needs to evaluate the formula using the assignments in $\tilde{\sigma}$. If $\tilde{\sigma}$ is a solution, it is then added to $\Sigma^{k+1}_\sigma$.

Note that, unlike QuickSampler, in SMTSampler we do not use a strategy of eager generation of samples. Instead, we compute the sets $\Sigma^1_\sigma, \Sigma^2_\sigma, \ldots$ in order. We chose this approach because it allows an adaptive generation of samples based on accuracy. During the construction of $\Sigma^{k+1}_\sigma$ from $\Sigma^k_\sigma$, we keep statistics on which fraction $\alpha$ of the checked samples were valid. If this fraction is below a certain threshold $\alpha_{\text{min}}$, such as 0.1, we do not generate
CHAPTER 3. SAMPLING FROM SAT AND SMT CONSTRAINTS

\[ \Sigma_{\sigma}^{k+2} \] and instead just proceed to the next epoch. This adaptive generation of samples allows us to avoid trying out too many invalid samples.

All the samples which are ultimately output by SMTSampler are the ones in \( \bigcup_{0 \leq k \leq 6} \Sigma_{\sigma}^k \), where we define \( \Sigma_{\sigma}^0 = \{ \sigma \} \). Those are all solutions to the formula, as the ones which were produced by the combination function for \( 2 \leq k \leq 6 \) have been checked for validity. We have found that this adaptive generation of samples is essential in some SMT formulas to avoid the generation of large number of invalid samples. We always use valid solutions as arguments to the combination function, which enables it to generate valid solutions with high probability.

Computing the Base Solution

The initial base solution \( \sigma \) for each epoch is computed similarly to QuickSampler. We first generate a random assignment \( \sigma' \) by choosing values to the Boolean and bit-vector variables in the formula uniformly at random. We do not assign values to the arrays and uninterpreted functions in \( \sigma' \), because we do not know initially which indices will be relevant for those variables. After generating \( \sigma' \), we choose \( \sigma \) as a solution which is as close as possible to \( \sigma' \). This is done to explore as much of the solution space as possible, generating base solutions \( \sigma \) from different parts of the space.

The problem of finding a solution \( \sigma \) which is as close as possible to \( \sigma' \) can be encoded as a MAX-SMT optimization problem to be solved by the constraint solver. We add one hard constraint stating that the formula \( \phi \) must be satisfied. For each bit-vector variable \( v \), we add one soft constraint \( v = \sigma'[v] \) stating that the \( v \) should have the same value that it had in \( \sigma' \). Analogously, we add one soft constraint \( b = \sigma'[b] \) for each Boolean variable \( b \).

Computing Atomic Mutations

After generating a base solution \( \sigma \), we first compute a set of neighboring solutions of the base solution \( \sigma \), before combining mutations to generate new samples. This is a different strategy from QuickSampler, which would eagerly combine mutations as new neighboring solutions are computed. In function computeNeighboringSolutions, the first step is collecting the set of conditions \( C_\sigma \) which are true for \( \sigma \). Then, MAX-SMT queries are used to produce new neighboring solutions. Each MAX-SMT query attempts to flip one condition, while maintaining the remaining conditions valid, if possible. We specify a maximum time budget allowed for this phase, such as 20 minutes, which is one third of the total time budget of one hour. If the time budget is enough to solve queries flipping each of the conditions in \( C_\sigma \), then all those queries will be made. Otherwise, we select randomly and uniformly a maximum subset of the conditions to be flipped and solved in MAX-SMT queries within the time limit. This is also essential for the large and complex SMT formulas handled by SMTSampler. When the number of variables hits hundreds or thousands, it often becomes infeasible to try to flip the values of each of them individually in MAX-SMT queries.
Table 3.1: Example of SMT conditions to be flipped.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b \in \text{Bool}$</td>
<td>$b = \text{False}$</td>
</tr>
<tr>
<td>$v \in \text{BV}$</td>
<td>$\text{extract}(v, 5) = 1$</td>
</tr>
<tr>
<td>$a \in \text{Array}$</td>
<td>$\text{extract}(a[011], 3) = 1$</td>
</tr>
<tr>
<td>$f \in \text{UF}$</td>
<td>$\text{extract}(f(0, 10), 1) = 0$</td>
</tr>
</tbody>
</table>

Constructing $C_\sigma$. Function getConditions produces $C_\sigma$ by collecting conditions for each variable in the formula. Table 3.1 shows one example condition for each of the variables types. Those are conditions that are valid for the example values from Table 2.1. Here, extract is a function that takes a bit-vector $v$ and an integer index $i$ and returns the value of the bit at index $i$ in $v$.

The conditions are generated as follows. For each Boolean variable, we add one condition $b = \sigma[b]$ asserting that the variable has the same value as in the base solution. For each bit-vector variable, we add one condition for each of its bits. The condition is of the form $\text{extract}(v, i) = \text{extract}(\sigma[v], i)$, asserting that, when extracting the given bit from the bit-vector, we obtain the same value that would be obtained from the base solution.

For each array $a$, we look at each of the indices $I(\sigma[a])$ assigned in the concrete instance of the array $\sigma[a]$. For each such index $x$, we consider the concrete bit-vector $\sigma[a][x]$ returned by the array on such index, and we add one condition for each bit in this bit-vector, such as $\text{extract}(a[x], i) = \text{extract}(\sigma[a][x], i)$. The procedure for uninterpreted functions is analogous. For each argument tuple that is assigned a value in the base solution, we recursively add conditions according to the value type.

Computing $\Sigma^1_\sigma$. Just like in QuickSampler, we compute neighboring solutions by picking one condition $c \in C_\sigma$ and using a MAX-SMT solver call MAX-SMT($\{\phi, \neg c\}, C_\sigma \setminus \{c\}$) to flip this condition, while keeping the remaining conditions as soft constraints.

One challenge in solving the MAX-SMT optimization problems is that they are expensive when the number of soft constraints is too large. For this reason, as an alternative, SMTSampler also allows the strategy of specifying only one soft constraint per bit-vector variable, instead of one for each bit in a bit-vector. For example, one would specify one condition as $v = 00100111$, instead of 8 different conditions such as $\text{extract}(v, 0) = 0$ and $\text{extract}(v, 1) = 0$. We evaluate this strategy in addition to our original strategy in Section 5.2. This alternative approach only changes the soft constraints that are added to the MAX-SMT query. For the hard constraint $\neg c$, we chose to always use conditions on the individual bits of each bit-vector, because we found that this is important to generate a larger number of atomic mutations and consequently a larger number of samples.
v: 1 1 0 0 0 1 0 1
\( \delta_a = v \oplus v_a: 1 0 0 0 1 1 0 \)
v_a: 0 1 0 0 0 0 1 1
\( \delta_b = v \oplus v_b: 0 0 0 1 0 1 0 0 \)
v_b: 1 1 0 1 0 0 0 1
\( (\delta_a \lor \delta_b): 1 0 0 1 0 1 1 0 \)
\( \psi_v(v_a, v_b) = v \oplus (\delta_a \lor \delta_b): 0 1 0 1 0 0 1 1 \)

Figure 3.3: Combining two mutations over \( v \in BV[8] \).

Combining Mutations

Now we define the combination function \( \Psi \) which is used to generate new samples. Assume that we already know the base solution \( \sigma \) and two additional solutions to the formula \( \sigma_a \) and \( \sigma_b \), which are close to \( \sigma \). Those additional solutions can be obtained by calling `computeNeighboringSolutions` or they could be already generated by an application of the \( \Psi \) function.

The combination function \( \Psi \), which combines entire solutions, is constructed by defining a method \( \psi \) to combine the values of each of the variables in the formula. We define

\[
\Psi_\sigma(\sigma_a, \sigma_b)[v] = \psi_{\sigma[v]}(\sigma_a[v], \sigma_b[v])
\]

This means that, in order to produce the assignment \( \Psi_\sigma(\sigma_a, \sigma_b) \), we simply use \( \psi \) to combine the assignments for each variable \( v \in Vars[\phi] \).

Next, we define how the combination method \( \psi \) is applied to each of the variable types. We first present the procedure for bit-vector variables and then generalize it to the other types. Let \( u \in BV \) be a bit-vector variable in the formula. We use the notations \( v, v_a, v_b \) to represent the values assigned to variable \( u \) in each of the solutions \( \sigma, \sigma_a, \sigma_b \), i.e., we define

\[
v = \sigma[u], v_a = \sigma_a[u], v_b = \sigma_b[u].
\]

Consider the bit-vectors presented in Figure 3.3. Just like in `QUICKSAMPLER`, we define the differences \( \delta_a = v \oplus v_a \) and \( \delta_b = v \oplus v_b \) computed by a bit-wise XOR. Those differences \( \delta_a \) and \( \delta_b \) indicate exactly which bits differ between the base value and each of the additional values. One can think of those differences as mutations, which can be applied to the base value in order to produce a different value. For example, we can compute \( v_a \) as \( v \oplus \delta_a \), where the XOR operator is used to apply mutation \( \delta_a \) to \( v \). For bit-vectors, we define a combined mutation through the OR operator, producing \( (\delta_a \lor \delta_b) \). This resulting mutation can be applied to the base value \( v \), producing a new value \( v \oplus (\delta_a \lor \delta_b) \). Thus, for bit-vectors, \( \psi \) is defined as

\[
\psi_v(v_a, v_b) = v \oplus ((v \oplus v_a) \lor (v \oplus v_b))
\]
For Boolean values, we use the same technique: $\psi_b(b_a, b_b) = b \oplus ((b \oplus b_a) \lor (b \oplus b_b))$. This way, Boolean values behave the same as bit-vectors of size 1.

Now we define how to apply the combination method $\psi$ to a base array $\alpha = \sigma[a]$ and two neighboring arrays $\alpha_a = \sigma_a[a]$ and $\alpha_b = \sigma_b[a]$. Remember that our array models only define explicit values for a finite set of indices. Assume that array $\alpha$ has explicitly defined values for indices in the set $I(\alpha)$, and a default value $d(\alpha)$ for all other indices. Arrays $\alpha_a$ and $\alpha_b$ are constructed analogously, with possibly different sets of assigned indices $I(\alpha_a)$ and $I(\alpha_b)$. We define the combination function for arrays as

$$\psi_\alpha(\alpha_a, \alpha_b)[x] = \begin{cases} \psi_{\alpha[x]}(\alpha_a[x], \alpha_b[x]), & \text{if } x \in I(\alpha) \cup I(\alpha_a) \cup I(\alpha_b) \\ \psi_{d(\alpha)}(d(\alpha_a), d(\alpha_b)), & \text{otherwise} \end{cases}$$

This means that the assigned indices of the generated array will be $I = I(\alpha) \cup I(\alpha_a) \cup I(\alpha_b)$, the union of the assigned indices of each of the three arrays. If $x \in I$, then $x$ may or may not have a non-default value assigned for each of the three arrays, while if $x \notin I$, we know that $x$ has a default value assigned for all the arrays. This definition keeps the generated array model $\psi_\alpha(\alpha_a, \alpha_b)$ simple, with explicitly defined values only for the set of indices $I$. For uninterpreted functions, the combination function is defined analogously, with the set of assigned argument tuples being the union of the assigned tuples for the base solution and the two neighboring solutions. This completes the definition of $\psi$ and, consequently, $\Psi$.

The definition of $\Psi_\sigma(\sigma_a, \sigma_b)$ is a natural extension from the original QUICKSAMPLER case, which only applied to Boolean variables. It obtains the mutation that generates $\sigma_a$ from $\sigma$ and the mutation that generates $\sigma_b$ from $\sigma$ and then combines those two mutations in an additive way. If $\sigma_a$ and $\sigma_b$ are neighboring solutions obtained from a MAX-SMT query, those mutations are atomic mutations, which represent a minimal set of bits that can be flipped and still preserve the satisfiability of the formula. Therefore, it is likely that there exist some clauses in the formula which establish a strong dependence between those bits. Since in the resulting sample we flip the bits which were flipped by either of the two atomic mutations, it is likely that such clauses would still be satisfied. Our experiments demonstrate that this combination of mutations is effective at generating diverse solutions, not only for SAT formulas, but also for such complex SMT formulas.

We note that the combination functions are commutative and associative with respect to the atomic mutations applied. More explicitly, we have $\Psi_\sigma(\sigma_a, \sigma_b) = \Psi_\sigma(\sigma_b, \sigma_a)$ and $\Psi_\sigma(\Psi_\sigma(\sigma_a, \sigma_b), \sigma_c) = \Psi_\sigma(\sigma_a, \Psi_\sigma(\sigma_b, \sigma_c))$. In SMTSampler, the samples generated in $\Sigma_\sigma^k$ can be seen as the combination of $k$ atomic mutations.
Chapter 4

Coverage-guided Sampling

In this chapter we formalize the problem of coverage-guided sampling and introduce our technique GUIDEDSAMPLER [25], which is an extension of SMTSAMPLER (presented in Section 3.2) designed to tackle this problem.

4.1 Formulation of Coverage-guided Sampling

In the following definitions, let $\phi$ be a satisfiable SMT formula (i.e., $Sols[\phi] \neq \emptyset$) over variables $V = \text{Vars}[\phi]$ and let $\psi_1, \psi_2, \ldots, \psi_n$ be $n$ SMT formulas which only include variables from $V$ (i.e., $\text{Vars}[\psi_i] \subseteq V$). The formulas $\psi_i$ represent the coverage predicates of interest.

**Definition 4.1.1. Coverage class.** Each vector of Boolean values $b = (b_1, b_2, \ldots, b_n) \in \mathbb{B}^n$ defines one coverage class $G_b$ of formula $\phi$ over the coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$, as follows:

$$G_b = \{ \sigma \in Sols[\phi] \mid \psi_i[\sigma] = b_i, \ i \in \{1, 2, \ldots, n\} \}$$

That is, the set of solutions to $\phi$ that satisfy $\psi_i[\sigma] = b_i$ for each predicate defines a class of solutions of $\phi$. Some of those classes might be empty, if they include no solutions to $\phi$. Let $N$ be the number of non-empty coverage classes, $N = |\{b \in \mathbb{B}^n \mid G_b \neq \emptyset\}|$. For each assignment $\sigma$ to the variables of $\phi$, its coverage class is defined as

$$G[\sigma] = G(\psi_1[\sigma], \psi_2[\sigma], \ldots, \psi_n[\sigma])$$

For our example formula $\phi = b \lor (x + 2 > y)$, if we have coverage predicates $\psi_1 = b$ and $\psi_2 = x + 2 > y$, then $G(\text{False, False}) = \emptyset$ and there are only $N = 3$ non-empty coverage classes.

**Definition 4.1.2. Ideal coverage-guided sampler.** An ideal coverage-guided sampler for formula $\phi$ with coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$ is a random process that returns each possible solution $\sigma \in Sols[\phi]$ with probability

$$P(\sigma) = \frac{1}{N \cdot |G[\sigma]|}$$
What this means is that we want to give equal weight to each of the non-empty coverage classes. Every non-empty coverage class should have the same probability $1/N$ of being sampled. Within each individual coverage class, all solutions should be sampled uniformly.

If we could enumerate all solutions to the constraint $\phi$, an ideal coverage-guided sampler could be constructed as follows.

1. First, pick one non-empty coverage class uniformly.
2. Then, uniformly pick one solution from that class.

However, for most practical problems, the number of solutions and coverage classes is astronomically large. Our goal is to devise an efficient algorithm that can approximate an ideal coverage-guided sampler.

4.2 GuidedSampler Algorithm

Just like SMTSampler, GuidedSampler also uses the core idea of computing mutations that can be applied to one solution in order to generate other solutions to the formula. We again define atomic mutations as the minimal changes between one solution and another neighboring solution to the formula. We similarly define a combination function which can be used to merge the effects of several distinct atomic mutations. This function generates a compound mutation and applies it to the original solution, producing a possibly new solution. While the initial computation of atomic mutations requires expensive solver calls, the combination function does not, making it very efficient. This efficient combination of solutions can be leveraged to generate millions of samples from just a few hundreds of atomic mutations. The samples generated by the combination function are assignments which satisfy the formula with high probability, even on large and complex industrial benchmarks.

The main new contribution of GuidedSampler is that we explicitly try to generate new solutions that belong to different coverage classes. This is achieved by three new modifications to the SMTSampler algorithm that enable it to reach new classes of solutions and also avoid sampling from the same repeated classes. Those modifications allow GuidedSampler to implement coverage-guided sampling, where the distribution of solutions generated is guided by the coverage predicates specified by the user.

Next we present the full details of the algorithm. First, we describe the main sampling procedure of GuidedSampler. Next, we explain how one base solution is chosen for each epoch, and how we discover a set of neighboring solutions to the base solution. Finally, we describe how those neighboring solutions are used to generate new samples.

Main GuidedSampler Algorithm

Algorithm 3 presents the main GuidedSampler procedure, which takes as input an SMT formula $\phi$ and $n$ coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$. The main algorithm is based on our prior tool SMTSampler. However, we added 3 important modifications to this base technique.
Algorithm 3 \textbf{GuidedSampler}(\phi, \{\psi_1, \psi_2, \ldots, \psi_n\}).

1: \textbf{function} GuidedSampler
2: \hspace{1em} \textbf{while not} done \textbf{do}
3: \hspace{2em} \( (b_1, b_2, \ldots, b_n) \leftarrow \text{generateRandomClass}(n) \)
4: \hspace{2em} \( S_b \leftarrow \{\psi_1 = b_1, \psi_2 = b_2, \ldots, \psi_n = b_n\} \)
5: \hspace{2em} \( \sigma' \leftarrow \text{generateRandomAssignment}(\phi) \)
6: \hspace{2em} \( C_{\sigma'} \leftarrow \text{getConditions}(\phi, \sigma') \)
7: \hspace{2em} \( \sigma \leftarrow \text{MAX-SMT}(\{\phi\}, S_b \cup C_{\sigma'}) \)
8: \hspace{2em} \text{output}(\{\sigma\})
9: \hspace{2em} \( (\Sigma_1^1, \Gamma_\sigma) \leftarrow \text{computeNeighboringSolutions}(\sigma) \)
10: \hspace{2em} \text{output}(\Sigma_1^1)
11: \hspace{2em} \( \alpha \leftarrow 1, k \leftarrow 1, \Sigma_{\sigma} \leftarrow \Sigma_1^1 \)
12: \hspace{2em} \textbf{while} \( \alpha \geq \alpha_{\text{min}} \land k < 6 \) \textbf{do}
13: \hspace{3em} \( (\Sigma_{k+1}^1, \alpha, \Sigma_{\sigma}, \Gamma_\sigma) \leftarrow \text{combineMutations}(\Sigma_k^1, \Sigma_1^1, \Sigma_{\sigma}, \Gamma_\sigma) \)
14: \hspace{2em} \text{output}(\Sigma_{k+1}^1)
15: \hspace{2em} \( k \leftarrow k + 1 \)
16: \textbf{function} computeNeighboringSolutions(\sigma)
17: \hspace{2em} \( (b_1, b_2, \ldots, b_n) \leftarrow \{\psi_1[\sigma], \psi_2[\sigma], \ldots, \psi_n[\sigma]\} \)
18: \hspace{2em} \( S_b \leftarrow \{\psi_1 = b_1, \psi_2 = b_2, \ldots, \psi_n = b_n\} \)
19: \hspace{2em} \( C_{\sigma} \leftarrow \text{getConditions}(\phi, \sigma) \)
20: \hspace{2em} \( \Sigma_1^1 \leftarrow \{\}, \Gamma_\sigma \leftarrow \{\} \)
21: \hspace{2em} \textbf{for} \ c \ \text{in} \ S_b \ \textbf{do}
22: \hspace{3em} \( \tilde{\sigma} \leftarrow \text{MAX-SMT}(\{\phi, \neg c\}, C_{\sigma} \cup S_b \setminus \{c\}) \)
23: \hspace{3em} \textbf{if} \ \tilde{\sigma} \notin \Sigma_{\sigma} \textbf{ then}
24: \hspace{4em} \( \Gamma_\sigma \leftarrow \Gamma_\sigma \cup G[\tilde{\sigma}] \)
25: \hspace{4em} \( \Sigma_{\sigma} \leftarrow \Sigma_{\sigma} \cup \{\tilde{\sigma}\} \)
26: \hspace{2em} \textbf{return} \ (\Sigma_1^1, \Gamma_\sigma) \)
27: \textbf{function} combineMutations(\Sigma_k^1, \Sigma_1^1, \Sigma_{\sigma}, \Gamma_\sigma)
28: \hspace{2em} \text{valid} \leftarrow 0, \text{checks} \leftarrow 0
29: \hspace{2em} \textbf{for} \ (\sigma_a, \sigma_b) \ \text{in} \ \Sigma_k^1 \times \Sigma_1^1 \ \textbf{do}
30: \hspace{3em} \( \tilde{\sigma} \leftarrow \Psi_{\sigma}(\sigma_a, \sigma_b) \)
31: \hspace{3em} \textbf{if} \ \tilde{\sigma} \notin \Sigma_{\sigma} \textbf{ then}
32: \hspace{4em} \( \Sigma_{\sigma} \leftarrow \Sigma_{\sigma} \cup \{\tilde{\sigma}\} \)
33: \hspace{4em} \text{checks} \leftarrow \text{checks} + 1
34: \hspace{3em} \textbf{if} \ \phi[\tilde{\sigma}] \land G[\tilde{\sigma}] \notin \Gamma_\sigma \textbf{ then}
35: \hspace{4em} \( \Gamma_\sigma \leftarrow \Gamma_\sigma \cup G[\tilde{\sigma}] \)
36: \hspace{4em} \( \Sigma_{k+1}^1 \leftarrow \Sigma_{k+1}^1 \cup \{\tilde{\sigma}\} \)
37: \hspace{2em} \text{valid} \leftarrow \text{valid} + 1
38: \hspace{2em} \textbf{return} \ (\Sigma_{k+1}^1, \text{valid}/\text{checks}, \Sigma_{\sigma}, \Gamma_\sigma) \)
CHAPTER 4. COVERAGE-GUIDED SAMPLING

31
to allow coverage-guided sampling based on the coverage predicates (changes are underlined in Algorithm 3):

M1 Generate neighboring solutions from a neighboring coverage class, by flipping the values of coverage predicates in function computeNeighboringSolutions.

M2 Discard solutions that belong to a previously seen coverage class in the current epoch (lines 23 and 34).

M3 Randomize the coverage class of the initial base solution \( \sigma \) (lines 3-7), instead of generating \( \sigma \) based only on the random assignment \( \sigma' \).

Next, we explain the algorithm in detail. The GuidedSampler algorithm works over several epochs. In each epoch, we start by generating an initial random solution \( \sigma \) to the formula, which we call a base solution. This is done by generating a random coverage class \( G_b \), as well as a random assignment \( \sigma' \) to the variables of the formula in lines 3-6 and using a MAX-SMT [53] solver call to obtain a solution \( \sigma \) which is closest to \( G_b \) and to \( \sigma' \) in line 7. The details of this procedure will be presented later. Then, in line 9, we use the function computeNeighboringSolutions to compute a set \( \Sigma^1_\sigma \) of neighboring solutions to \( \sigma \) that belong to different coverage classes than \( \sigma \). This function will also be described later.

For combining mutations, we leverage the same combination function \( \Psi \) constructed in Section 3.2 for SMTSampler. Again, the combination function is defined as \( \Psi : X \times X \times X \rightarrow X \), where \( X \) is the space of all possible assignments to the variables in \( V = \text{Vars}[\phi] \). We denote by \( \Psi_\sigma(\sigma_a, \sigma_b) \) the application of the combination function to the base solution \( \sigma \) and two other solutions \( \sigma_a \) and \( \sigma_b \). Intuitively, the combination function \( \Psi \) computes the mutations which can be applied to \( \sigma \) to generate \( \sigma_a \) and \( \sigma_b \), then merges those two mutations together to produce a new assignment. The assignment returned by \( \Psi \) is not guaranteed to satisfy the formula, but in practice it is a solution with high probability. This is because the atomic mutations capture the minimal changes that preserve the satisfiability of the formula, and we designed \( \Psi \) to combine those changes in an additive way.

We next describe how function combineMutations called by GuidedSampler in line 13 uses \( \Psi \) to generate new samples. We denote by \( \Sigma^1_\sigma \) the set of neighboring solutions to \( \sigma \) obtained from computeNeighboringSolutions. Starting from \( \Sigma^1_\sigma \), our goal is to compute sets \( \Sigma^k_\sigma \) which will contain solutions generated by combining \( k \) atomic mutations, for \( 1 \leq k \leq 6 \). Throughout the current epoch, we maintain a set \( \Sigma_\sigma \) of samples which were computed so far, both valid and invalid. Initially, \( \Sigma_\sigma = \Sigma^1_\sigma \). We also maintain a set \( \Gamma_\sigma \) of all the coverage classes for which we have generated a solution in the current epoch.

Now assume that we already constructed a set \( \Sigma^k_\sigma \). We can inductively build the set \( \Sigma^{k+1}_\sigma \) as follows. For each pair of samples \( \sigma_a \in \Sigma^k_\sigma \) and \( \sigma_b \in \Sigma^1_\sigma \), we apply the combination function \( \Psi \) to generate a new sample \( \tilde{\sigma} = \Psi_\sigma(\sigma_a, \sigma_b) \). If \( \tilde{\sigma} \) is an element of \( \Sigma_\sigma \), it has already been checked and is discarded. Otherwise, we add it to \( \Sigma_\sigma \) and check if it is a solution to the formula. Following our modification M2, we also check if its coverage class \( G[\tilde{\sigma}] \) is one for which we have already found a solution in the current epoch. Those checks are fast, as
they only need to evaluate the formulas using the assignments in $\tilde{\sigma}$. If $\tilde{\sigma}$ is a solution from a previously unseen class, it is then added to $\Sigma^{k+1}_{\sigma}$.

Just like SMTSampler, during the construction of $\Sigma^{k+1}_{\sigma}$ from $\Sigma^{k}_{\sigma}$, we also keep statistics on which fraction $\alpha$ of the checked samples were valid. This allows stopping the combination earlier when we detect that the algorithm is generating too many invalid samples. The samples ultimately output by GuidedSampler are the ones in $\bigcup_{0 \leq k \leq 6} \Sigma^{k}_{\sigma}$, where we define $\Sigma^{0}_{\sigma} = \{\sigma\}$. Those are all solutions to the formula, as the ones which were produced by the combination function for $2 \leq k \leq 6$ have been checked for validity.

**Computing the Base Solution**

Now, we describe how the initial base solution $\sigma$ is obtained. We first generate a random coverage class $G_b$ by picking $n$ random Boolean values $b = (b_1, b_2, \ldots, b_n)$. We also generate a random assignment $\sigma'$ by choosing values to the Boolean and bit-vector variables in the formula uniformly at random. After choosing $G_b$ and $\sigma'$, we want to find a solution $\sigma$ which is as close as possible to the coverage class $G_b$ and to the random assignment $\sigma'$. This is done to explore as much of the coverage classes and the solution space as possible.

The problem of finding a solution $\sigma$ which is as close as possible to $G_b$ and $\sigma'$ can be encoded as a MAX-SMT [53] optimization problem to be solved by the constraint solver. We add one hard constraint stating that the formula $\phi$ must be satisfied. We also add two sets of soft constraints, $S_b$ and $C_{\sigma'}$. For each coverage predicate, we add one soft constraint to $S_b$ saying that the predicate $\psi_i$ should be equal to the random Boolean $b_i$. For each bit-vector or Boolean variable $v \in Vars[\phi]$, we add one soft constraint to $C_{\sigma'}$ stating that variable $v$ should have the same value that it has in $\sigma'$.

**Computing Atomic Mutations**

After generating a base solution $\sigma$, we compute neighboring solutions of the base solution $\sigma$, so that their atomic mutations can be combined to generate new samples. When doing so, we want to generate neighboring solutions that are from different classes than the base solution $\sigma$. We want our neighboring solutions to flip the value of some of the coverage predicates. The first step is collecting a set of conditions $S_b$ which are true about the coverage predicates for the base solution $\sigma$. Then, MAX-SMT queries are used to produce new neighboring solutions. Each MAX-SMT query attempts to flip one of those conditions, while maintaining the remaining conditions valid, if possible. We also add soft constraints for the variables in $\phi$, saying that those variables should keep the same value they had in the base solution, if possible. This is done because we want the neighboring solution to be as close as possible to the original base solution, so that the atomic mutations are small and simple. We found that this is essential for having a good probability of generating valid samples when combining mutations.
Constructing $S_b$. The definition of $S_b$ is the same that we presented before, for the computation of the base solution. For each coverage predicate, we add one condition to $S_b$ saying that this predicate has the same value it had in the base solution $\sigma$. Then, in our MAX-SMT queries, we will try to flip one of those conditions, while maintaining as many as possible of the other conditions in $S_b$ true. The goal is that the neighboring solution produced should come from a neighboring coverage class, that only flips a minimal number of coverage predicates.

Constructing $C_\sigma$. Function getConditions produces $C_\sigma$ by collecting conditions for each variable in the formula. The conditions can be, for example, \text{extract}(v,5) = 1, saying that the bit of index 5 in the bit-vector $v$ has value 1. Those are the same conditions generated by SMTSampler using its default strategy (SMTbit). For each bit inside each variable in the formula, we add a condition asserting that this bit has the same value as in the base solution.

Computing $\Sigma^1_\phi$. After collecting the conditions in $S_b$ and $C_\sigma$, we want to compute neighboring solutions by picking one condition $c \in S_b$ and using the constraint solver to find a solution to $\phi \land \neg c$. However, the neighboring solution should be as similar as possible to $\sigma$. We express such constraint by requiring that the new solution should satisfy the maximum possible number of the remaining conditions in $S_b \setminus \{c\}$ and $C_\sigma$. Those requirements can be specified as a MAX-SMT optimization problem. We specify two hard constraints $\{\phi, \neg c\}$, stating that we want a solution to $\phi$ that does not satisfy $c$. And we also specify as soft constraints the conditions in $S_b \setminus \{c\}$ and in $C_\sigma$.

Combining Mutations

The combination function $\Psi$ which is used to produce new samples is the same as in SMTSampler. Our experiments demonstrate that this combination of mutations is effective at generating valid samples, even for large and complex SMT formulas. We also found that the coverage predicates tend to follow the same pattern. That is, when we compute the value of a coverage predicate $\psi_i$ on the resulting sample, most of the time it will be the same value that would result from combining mutations of the predicate values. Figure 4.1 shows an example of the combination method used to combine mutations over a bit-vector variable of 8 bits. As shown in the last 3 columns of Figure 4.1, the predicates in this case follow the same rule. This helps our algorithm generate solutions from new diverse coverage classes.
Figure 4.1: Combining two mutations over bit-vectors of size 8. We also list values for 3 coverage predicates $\psi_1, \psi_2, \psi_3$. 

<table>
<thead>
<tr>
<th>Bits</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$ :</td>
<td>1 1 0 0 0 1 0 1</td>
</tr>
<tr>
<td>$v_a$ :</td>
<td>0 1 0 0 0 1 1</td>
</tr>
<tr>
<td>$v_b$ :</td>
<td>1 1 0 1 0 0 0 1</td>
</tr>
<tr>
<td>$\delta_a = v \oplus v_a$ :</td>
<td>1 0 0 0 0 1 1 0</td>
</tr>
<tr>
<td>$\delta_b = v \oplus v_b$ :</td>
<td>0 0 0 1 0 1 0 0</td>
</tr>
<tr>
<td>$(\delta_a \lor \delta_b)$ :</td>
<td>1 0 0 1 0 1 1 0</td>
</tr>
<tr>
<td>$\psi_v(v_a, v_b) = v \oplus (\delta_a \lor \delta_b)$ :</td>
<td>0 1 0 1 0 0 1 1</td>
</tr>
</tbody>
</table>
Chapter 5
Evaluation

This chapter presents a detailed experimental evaluation of our new techniques. In Section 5.1, we evaluate QuickSampler in terms of the correctness of samples generated, its performance in comparison with previous state-of-the-art techniques and the uniformity of the solutions. In Section 5.2, we evaluate SMTSampler in comparison to the baseline approach of converting the formula into SAT, both in terms of the number of unique solutions generated and the coverage of the SMT formula. Finally, in Section 5.3, we evaluate GuidedSampler against SMTSampler and other baseline techniques, in terms of the number of unique coverage classes reached and the uniformity of solutions over the coverage classes.

5.1 QuickSampler Evaluation

We have implemented\(^1\) QuickSampler in C++, using Z3 [24] as the underlying solver. QuickSampler uses the Z3 optimization subsystem \(\nu Z\) [9] in order to solve the MAX-SAT queries. We also use the \texttt{push()} and \texttt{pop()} interfaces to efficiently add and remove constraints from a single solving context.

QuickSampler takes as input a SAT formula in conjunctive normal form (CNF), represented in the DIMACS file format. The formula includes a list of variables which compose its independent support.

Our implementation outputs the samples generated to a file without checking if they are valid solutions. If desirable, it is possible to add a posterior check which verifies if the samples are valid or not (and possibly filters out the invalid ones). We also do not check for duplicates, which can appear between different epochs in the sampling algorithm. This global check for uniqueness could also be added, but it would require an additional time and memory overhead. Some applications might prefer not to keep all generated samples in memory, and allow the generation of repeated samples instead.

\(^1\)The source code is available at \url{https://github.com/RafaelTupynamba/quicksampler}. 
We have implemented an offline analysis to check if the samples are valid and generate histograms that count how many times each solution has been sampled. We record the time taken by the sample generation and also the time taken by the checking phase. The checking phase is not heavily optimized and for most benchmarks it was more expensive than the sampling phase. We believe there is still room for improvement in the checking phase, since all it needs to do is to propagate the values of the independent support to the remaining variables and check if all clauses are satisfied.

Experiments

We evaluate QuickSampler by comparing against two state-of-the-art samplers, namely UniGen2 [15] and SearchTreeSampler [29]. UniGen2 provides strong uniformity guarantees, by using hash functions composed of XOR constraints in order to partition the search space into similar-sized bins.

SearchTreeSampler, on the other hand, uses the SAT solver to find pseudosolutions (partial assignments to the first few variables) and progressively augments the pseudosolutions into real solutions. SearchTreeSampler is only approximately uniform. The uniformity can be increased with a higher number of samples per level $k$, but at a cost of also increasing the number of solver calls required. In our experiments, we used the default value of $k = 50$.

Both QuickSampler and UniGen2 can leverage the knowledge of an independent support of the formula to improve sampling performance. So in order to make for a fair comparison, we modified SearchTreeSampler to use this additional information. We reorder the variables of the formula in order to place first the ones which are part of the independent support. And we additionally tell SearchTreeSampler to finish sampling after processing those variables and output pseudosolutions (assignments to the variables of the independent support) that it has produced so far. Since an assignment to the independent support can only be completed to one solution, there is no need to find assignments to the remaining variables.

For the evaluation, we use the set of benchmarks from the UniGen2 paper [15] available online. From the benchmarks listed in [15], we found 173 on the online repository. The benchmarks include bit-blasted versions of SMT-LIB benchmarks, ISCAS89 circuits augmented with parity conditions on randomly chosen subsets of outputs and next-state variables, problems arising from automated program synthesis and constraints arising in bounded model checking. Thus, they are representative of the kinds of constraints that might appear in SMT formulas for software testing or circuit constraints for hardware.

On 10 benchmarks, UniGen2 reported an error because the specified independent sup-

---

2 Benchmarks and source code for UniGen2 were obtained from https://bitbucket.org/kuldeepmes1/unigen.

CHAPTER 5. EVALUATION

Table 5.1: Correctness statistics for the samples produced in one epoch of QuickSampler (average among all benchmarks).

<table>
<thead>
<tr>
<th>Atomic mutations</th>
<th>Total</th>
<th>Valid</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>32</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>511</td>
<td>492</td>
<td>96%</td>
</tr>
<tr>
<td>3</td>
<td>5619</td>
<td>5208</td>
<td>93%</td>
</tr>
<tr>
<td>4</td>
<td>47493</td>
<td>42179</td>
<td>89%</td>
</tr>
<tr>
<td>5</td>
<td>34637</td>
<td>282860</td>
<td>82%</td>
</tr>
<tr>
<td>6</td>
<td>2143385</td>
<td>1571553</td>
<td>73%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2543409</td>
<td>1902325</td>
<td>75%</td>
</tr>
</tbody>
</table>

The experiments were conducted on a 12-core, 3.50GHz Intel Core i7-5930K CPU. For each benchmark, each of the algorithms was given one core and 1.5 GB of memory. For QuickSampler and SearchTreeSampler, we allowed a maximum timeout of 1 hour, or 2 hours on the hardest benchmarks. We also stopped the sampling after a large number of samples had been produced (at least 10 million samples).

For UniGen2, we requested the generation of 1000 samples for most benchmarks, allowing up to 20 hours to produce them. On the hardest benchmarks, we reduced the number of requested samples to 500. For all the benchmarks in which UniGen2 failed to produce any samples, it times out after 20 hours even when the number of requested samples was 1.

Correctness of Samples

On Table 5.1, we list the average number of samples produced and how many of those were valid, on one epoch of the sampling algorithm. The results were averaged across all 163 benchmarks. They are classified according to the number of individual atomic mutations
CHAPTER 5. EVALUATION

which compose the mutation. The base solution used in the epoch is the one with 0 atomic mutations and the neighbors of the base solution obtained when flipping each bit correspond to 1 atomic mutation. Those are always valid solutions to the formula, since they are obtained as the result of solver calls.

From 2 to 6 atomic mutations, we see that the fraction of valid solutions decreases from 96% to 73%. And overall, 75% of all samples produced were valid, when we allow a maximum of 6 atomic mutations. Table 5.1 shows that, by adjusting this maximum, we can change the accuracy of the sampling. For example, with a maximum of 5 atomic mutations instead of 6, the fraction of valid samples would increase to 83%. However, there would be a substantial decrease in the quantity of samples produced. We have chosen to use the maximum number of 6 atomic mutations to allow the generation of millions of samples, while still having a reasonably good accuracy of 75%.

If \( n \) is the number of atomic mutations, the number of mutations of level 6 can go up to \( \binom{n}{6} \), a sixth-degree polynomial in \( n \). This explains why we can generate millions of samples from only tens of atomic mutations.

Performance Comparison

We define \( t_q, t_s, t_u \) to be the average times taken by \texttt{QuickSampler}, \texttt{SearchTreeSampler} and \texttt{UniGen2}, respectively to produce a valid sample. \( t_q \) was computed as \( t_q = T_q / (s_q \cdot p) \), where \( T_q \) is the total execution time, \( s_q \) is the total number of samples produced and \( p \) is the fraction of samples which are valid for \texttt{QuickSampler}. We additionally define \( t_q^* \) to be the estimated time per valid sample that \texttt{QuickSampler} would require if it also checked all samples for validity. \( t_q^* \) was computed as \( t_q^* = (T_q + T_c) / (s_q \cdot p) \), where \( T_c \) is the total time taken to check the validity of all \( s_q \) produced samples.

Table 5.2 shows the benchmarks used for the evaluation of \texttt{QuickSampler}. We have included the largest benchmarks (more than 4000 variables), the benchmarks which were listed as representative benchmarks in [15] and the benchmarks used for uniformity plots, which will be presented later. This includes the benchmarks which \texttt{QuickSampler} or \texttt{SearchTreeSampler} found hard.

The first group of columns in Table 5.2 shows basic information about the benchmarks: size of the independent support, number of variables, clauses and solutions. The number of solutions was obtained from \texttt{UniGen2}. On most benchmarks, an exact number of solutions is known, while for some we only know an approximation (represented with \( \approx \)) and on some \texttt{UniGen2} failed completely to compute the number of solutions. The second group of columns shows some results for \texttt{QuickSampler}: the number of epochs completed, number of MAX-SAT solver calls, number of samples generated and fraction of samples which are valid.

In Table 5.3, we show a performance comparison between the different techniques on the same set of benchmarks. For \texttt{QuickSampler}, we repeat the information about the number of samples generated and fraction of samples which are valid, and additionally include the
average times per valid sample $t_q$ and $t_q^*$, in microseconds. The third and fourth groups of columns present results for SearchTreeSampler and UniGen2: the number of samples produced and the average time per sample, taken in comparison with the QuickSampler time $t_q$.

The mean value for some ratios of interest is shown on Table 5.4. For example, $t_s/t_q \approx 10^{2.5\pm0.8}$. This was computed by taking the average and the standard deviation of log10($t_s/t_q$) across all benchmarks.

Figure 5.1 shows a comparison of the average time per valid sample on all benchmarks, against SearchTreeSampler and UniGen2. As reported in Table 5.4, QuickSampler was on average 2.5 orders of magnitude faster than SearchTreeSampler and 4.7 orders of magnitude faster than UniGen2. Overall, QuickSampler was only slower than SearchTreeSampler on the benchmark diagStencilClean.sk.41*36, with $t_s/t_q = 6.6 \times 10^{-5}$. We believe QuickSampler did not do well on diagStencilClean.sk.41*36 because the Z3 solver used in our implementation did not perform well on this formula. In comparison, MiniSAT, the solver used by SearchTreeSampler, was much faster on this benchmark.

| Benchmark         | $|S|$ | Vars | Clauses | Solutions | Epochs | QuickSampler Calls | Samples | Valid |
|-------------------|-----|------|---------|-----------|--------|---------------------|---------|-------|
| blasted_case47    | 28  | 118  | 328     | 262144    | 244    | 6616               | 10010929.564 |
| blasted_case110   | 17  | 287  | 1263    | 16384     | 1387   | 22208              | 10001202.822  |
| s820a.7.4         | 23  | 616  | 1703    | 591872    | 128    | 3093               | 10002673.770  |
| s820a.15.7        | 23  | 685  | 1987    | 722944    | 114    | 2579               | 10014350.674  |
| s1238a.3.2        | 32  | 686  | 1850    | 2466250752| 9      | 328                | 10140047.936  |
| s1196a.3.2        | 32  | 690  | 1805    | 1038090240| 11     | 393                | 10077447.803  |
| s832a.15.7        | 23  | 693  | 2017    | 3713024   | 83     | 2014               | 10017640.818  |
| blasted_case1.10b2| 45  | 827  | 2275    | 2748796944| 1      | 89                 | 10021799.739  |
| blasted_squaring16| 72  | 1627 | 5835    | 186527590882| 0      | 65                 | 10304220.209  |
| blasted_squaring7 | 72  | 1628 | 5837    | 274408144896| 0      | 68                 | 11349420.112  |
| 70.sk.3.40        | 40  | 4670 | 15864   | 8589934592 | 8      | 304                | 10134785.1000 |
| ProcessBean.sk.8.64| 64 | 4768 | 14458   | $\approx700936627072$ | 1      | 86                 | 10011221.906  |
| 56.sk.6.38        | 38  | 4842 | 17828   | 3690987520 | 9      | 334                | 10049283.930  |
| 35.sk.3.52        | 52  | 4915 | 10547   | 439804651104| 2      | 95                 | 10717156.1000 |
| 80.sk.2.48        | 48  | 4969 | 17060   | 109951162776| 2      | 126                | 10252598.1000 |
| 7.sk.4.50         | 50  | 6683 | 24816   | 219902325552| 2      | 124                | 10139607.1000 |
| doublyLinkedList.sk.8.37 | 37 | 6890 | 26918   | 2038431744 | 106    | 3425               | 10003513.267  |
| 19.sk.3.48        | 48  | 6993 | 23867   | 2959802892288| 1      | 89                 | 10198861.937  |
| 29.sk.3.45        | 45  | 8866 | 31557   | 347892350976| 2      | 120                | 10045170.855  |
| isolateRightmost.sk.7.481 | 481 | 10057 | 35275 | - | 0 | 59 | 11191269.878 | |
| 17.sk.3.45        | 45  | 10090 | 27056 | 27487969644 | 3 | 157 | 10181716.1000 | |
| 81.sk.5.51        | 51  | 10775 | 38006 | 18141941858304 | 1 | 52 | 11095985.867 | |
| LoginService2.sk.23*36 | 36 | 11511 | 41411 | $\approx163840$ | 272 | 6019 | 10001533.724 | |
| sort.sk.8.52      | 52  | 12125| 49611   | $\approx88046829568$ | 2 | 105 | 10563617.625 | |
| parity.sk.11.11*  | 11  | 13116| 47506   | - | 68 | 615 | 3833.809 | |
| 77.sk.3.44        | 44  | 14535| 27573   | 18253611008 | 6 | 249 | 10014904.966 | |
| 20.sk.1.51        | 51  | 15475| 60994   | 3710851743744 | 1 | 52 | 11126152.910 | |
| enqueueSeqSK.sk.10.42 | 42 | 16466 | 58515 | $\approx335443200$ | 4 | 207 | 10008980.762 | |
| karatsuba.sk.7.41* | 41 | 19594 | 82417 | $\approx1245184$ | 2 | 86 | 670641.088 | |
| diagStencilClean.sk.41*36* | 36 | 378131 | 2110471 | $\approx13$ | 5 | 66 | 87.701 | |
| tutorial3.sk.4.31* | 31 | 486193 | 2598178 | $\approx49283072$ | 6 | 193 | 2114947.798 | |

Table 5.2: Benchmarks for evaluation of QuickSampler.
Table 5.3: Comparison of SAT sampling algorithms.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Samples</th>
<th>QUICKSampler</th>
<th>SearchTreeSampler</th>
<th>UniGen2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$t_q$ ($\mu$s)</td>
<td>$t^*_q$ ($\mu$s)</td>
<td>$t_s/t_q$</td>
</tr>
<tr>
<td>blasted_case47</td>
<td>10010929</td>
<td>0.564</td>
<td>7.5</td>
<td>26</td>
</tr>
<tr>
<td>blasted_case110</td>
<td>10001202</td>
<td>0.822</td>
<td>28.3</td>
<td>29</td>
</tr>
<tr>
<td>s820a_7_4</td>
<td>10002673</td>
<td>0.770</td>
<td>5.9</td>
<td>34</td>
</tr>
<tr>
<td>s820a_15_7</td>
<td>10014350</td>
<td>0.674</td>
<td>9.0</td>
<td>66</td>
</tr>
<tr>
<td>s1728a_3_2</td>
<td>10140047</td>
<td>0.936</td>
<td>2.7</td>
<td>211</td>
</tr>
<tr>
<td>s1196a_3_2</td>
<td>10077447</td>
<td>0.803</td>
<td>3.2</td>
<td>246</td>
</tr>
<tr>
<td>s832a_15_7</td>
<td>1007640</td>
<td>0.818</td>
<td>6.4</td>
<td>100</td>
</tr>
<tr>
<td>blasted_case1_7_b12_2</td>
<td>10021799</td>
<td>0.739</td>
<td>2.9</td>
<td>305</td>
</tr>
<tr>
<td>blasted_squaring16</td>
<td>10304220</td>
<td>0.209</td>
<td>15.8</td>
<td>1961</td>
</tr>
<tr>
<td>blasted_squaring7</td>
<td>11344920</td>
<td>0.112</td>
<td>32.1</td>
<td>3788</td>
</tr>
<tr>
<td>ProcessBean.sk_8_64</td>
<td>10011221</td>
<td>0.906</td>
<td>4.1</td>
<td>1294</td>
</tr>
<tr>
<td>56.sk_6_38</td>
<td>10049283</td>
<td>0.930</td>
<td>5.3</td>
<td>694</td>
</tr>
<tr>
<td>35.sk_3_52</td>
<td>10717156</td>
<td>1.000</td>
<td>2.3</td>
<td>229</td>
</tr>
<tr>
<td>80.sk_2_48</td>
<td>10252958</td>
<td>1.000</td>
<td>4.0</td>
<td>1399</td>
</tr>
<tr>
<td>7.sk_4_50</td>
<td>10139607</td>
<td>1.000</td>
<td>4.9</td>
<td>1778</td>
</tr>
<tr>
<td>doublyLinkedListList.sk_8_37</td>
<td>10003513</td>
<td>0.267</td>
<td>678.4</td>
<td>6308</td>
</tr>
<tr>
<td>19.sk_3_48</td>
<td>10198861</td>
<td>0.937</td>
<td>4.1</td>
<td>2010</td>
</tr>
<tr>
<td>29.sk_3_45</td>
<td>10045170</td>
<td>0.855</td>
<td>6.7</td>
<td>2772</td>
</tr>
<tr>
<td>isolateRightmost.sk_7_481</td>
<td>11191269</td>
<td>0.878</td>
<td>11.3</td>
<td>3200</td>
</tr>
<tr>
<td>7.sk_4_50</td>
<td>10139607</td>
<td>1.000</td>
<td>4.9</td>
<td>1778</td>
</tr>
<tr>
<td>LoginService2.sk_23_36*</td>
<td>10001533</td>
<td>0.724</td>
<td>680.3</td>
<td>3212</td>
</tr>
<tr>
<td>sort.sk_8_52</td>
<td>10563617</td>
<td>0.625</td>
<td>31.1</td>
<td>7354</td>
</tr>
<tr>
<td>parity.sk_11_11*</td>
<td>3833</td>
<td>0.809</td>
<td>2322699.9</td>
<td>3535813</td>
</tr>
<tr>
<td>77.sk_3_44</td>
<td>10014004</td>
<td>0.966</td>
<td>5.8</td>
<td>1580</td>
</tr>
<tr>
<td>20.sk_1_51</td>
<td>11126152</td>
<td>0.910</td>
<td>4.0</td>
<td>3751</td>
</tr>
<tr>
<td>enqueueSeqSK.sk_10_42</td>
<td>10008980</td>
<td>0.762</td>
<td>34.8</td>
<td>21412</td>
</tr>
<tr>
<td>karatsuba.sk_7_41*</td>
<td>670641</td>
<td>0.088</td>
<td>125504.0</td>
<td>203615</td>
</tr>
<tr>
<td>diagStencilClean.sk_41_36*</td>
<td>87</td>
<td>0.701</td>
<td>120363466</td>
<td>120397476</td>
</tr>
<tr>
<td>tutorial3.sk_4_31*</td>
<td>2114947</td>
<td>0.798</td>
<td>4281.2</td>
<td>362747</td>
</tr>
</tbody>
</table>

The opposite effect can be seen, for example, on parity.sk_11_11, where MiniSAT was only able to complete a small number of solver calls.

Next we present graphs of the same metrics, but now also taking into account the time that would be required for QUICKSampler to check if the samples are valid. This should only be needed if the application cannot deal with invalid samples. Figures 5.2a and 5.2b show the comparison with SearchTreeSampler and UniGen2, respectively. We see that QUICKSampler is still 1 order of magnitude faster than SearchTreeSampler and 3.2 orders of magnitude faster than UniGen2, even when including this checking time. QUICKSampler was only slower than SearchTreeSampler on three benchmarks, where the ratio $t_s/t^*_q$ was 0.95 for 17.sk_3_45, 0.71 for 70.sk_3_40 and $6.6 \cdot 10^{-5}$ for diagStencilClean.sk_41_36.

Those results show clearly that QUICKSampler is capable of generating valid solutions orders of magnitude faster than the other techniques. However, we believe that an even more important metric is the number of unique valid solutions generated over time, since repeated solutions do not help uncover new behavior in the test program. So we performed
CHAPTER 5. EVALUATION

Figure 5.1: Average time per valid sample.

Table 5.4: Mean ratio comparisons across all benchmarks.

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_s/t_q)</td>
<td>(10^{2.5\pm0.8})</td>
</tr>
<tr>
<td>(t_u/t_q)</td>
<td>(10^{4.7\pm1.0})</td>
</tr>
<tr>
<td>(t_s/t^*)</td>
<td>(10^{1.0\pm0.5})</td>
</tr>
<tr>
<td>(t_u/t^*)</td>
<td>(10^{3.2\pm0.7})</td>
</tr>
<tr>
<td>(u_q/u_s)</td>
<td>(10^{2.3\pm0.7})</td>
</tr>
<tr>
<td>(u_q/u_u)</td>
<td>(10^{4.4\pm1.1})</td>
</tr>
</tbody>
</table>

an experiment to evaluate the number of unique valid solutions generated.

All three algorithms were allowed to run until they produced 10 million samples or reached 1 hour of execution. If their execution times are \(T_q, T_s, T_u\), we define \(T = \min\{T_q, T_s, T_u\}\) and look at the number of unique valid solutions that each algorithm could produce in time \(T\) and represent those numbers as \(u_q, u_s, u_u\). We found out that on most benchmarks \textsc{QuickSampler} was able to produce 10 million samples before 1 hour and it was the fastest algorithm to finish. So the uniqueness comparison is performed at time \(T_q\). On six benchmarks, neither of the algorithms could produce 10 million samples before 1 hour, so the uniqueness comparison is performed at 1 hour. The names of those benchmarks are marked with an asterisk in Table 5.3.

Figure 5.3a compares \textsc{QuickSampler} and \textsc{SearchTreeSampler} on the number of
Figure 5.2: Average time per valid sample, including time to check validity.

Figure 5.3: Unique solutions produced over same amount of time.
unique solutions produced. On average, QuickSampler produces 2.3 orders of magnitude more unique solutions, as seen in Table 5.4. On only one benchmark it was lower (karatsuba.sk.7.41, with \( u_q/u_s = 0.76 \)).

In Figure 5.3b, we present the ratio of unique solutions between QuickSampler and UniGen2. Again, the ratio was lower only on karatsuba.sk.7.41, with \( u_q/u_u = 0.08 \). On average, \( u_q \) was 4.4 orders of magnitude higher than \( u_u \). We found that QuickSampler performed poorly on karatsuba.sk.7.41 because it had not completed one sampling epoch within the first hour of execution, and most of the samples are generated towards the end of the sampling epoch. However, within 2 hours, QuickSampler was able to complete 2 sampling epochs, generating a vastly larger amount of samples, as reported in Table 5.3.

**Uniformity of Coverage**

The previous results show that QuickSampler can produce unique valid solutions very fast, which was our primary goal. But we would still like to check if the distribution of samples produced is similar to uniform, because we don’t want to be missing a large portion of the solution space, while focusing on a very biased subset of solutions. We have designed QuickSampler to start from a random point in the space of possible variable assignments in order to make our coverage more uniform. This also guarantees that any solution has a positive probability of being output by our algorithm.

In order to empirically evaluate the uniformity of QuickSampler, we compare its distribution of solutions with the ones from the two other samplers SearchTreeSampler, UniGen2 as well as a distribution from a perfect uniform sampler. Only the valid samples are considered in this analysis. We compare on the benchmarks for which the number of samples generated by UniGen2 in a time limit of 10 hours was at least five times the total number of solutions. It is important for statistical significance that each solution be sampled on average at least five times. For each of the benchmarks, let \( s_q, s_s, s_u \) be the number of valid samples generated by each algorithm and \( s = \min\{s_q, s_s, s_u\} \). We subsample uniformly \( s \) samples from the valid samples produced by each algorithm, and we also generate \( s \) samples from a perfectly uniform distribution, using the total number of solutions provided by UniGen2.

Figure 5.4 show the results of the comparison on all benchmarks for which the number of generated samples \( s \) can be at least five times the number of solutions before the timeout is reached. The \( x \) axis represents the number of times each solution has been sampled and the \( y \) axis represents the quantity of solutions which have been sampled \( x \) times. We can see that SearchTreeSampler and UniGen2 are usually indistinguishable from uniform, but QuickSampler is also very close to uniform behavior.

We have also applied Pearson’s chi-squared test to the \( s \) samples obtained from each algorithm. We compute the \( \chi^2 \) statistic and the corresponding p-value using the known number of solutions to the formula. Here, we reject the null hypothesis that the distribution is uniform if the p-value is lower than the confidence level of 0.05. This gives a bound on the type I error rate (i.e., the probability that a uniform distribution is mistakenly rejected as non-
Figure 5.4: Histograms comparing the distribution of solutions.

Table 5.5: Chi-squared uniformity test.

<table>
<thead>
<tr>
<th></th>
<th>Not Rejected</th>
<th>Rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUICKSAMPLER</td>
<td>149</td>
<td>11</td>
</tr>
<tr>
<td>SEARCHTREE_SAMPLER</td>
<td>153</td>
<td>7</td>
</tr>
<tr>
<td>UniGen2</td>
<td>155</td>
<td>5</td>
</tr>
</tbody>
</table>
uniform \(^4\). Table 5.5 show the results of applying this test to the 160 benchmarks for which we know an estimate of the number of solutions. We can see that SEARCHTREE_SAMPLER and UNI_GEN2 are more uniform, but QUICK_SAMPLER is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QUICK_SAMPLER completed a small number of epochs or when the number of produced samples is too low.

Besides analyzing the uniformity of the distribution, we also measured the number of unique valid solutions generated. This is arguably more important than the histograms of solution counts, because we want unique solutions to increase coverage in testing.

We computed the number \( u \) of unique valid solutions generated by QUICK_SAMPLER and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We record the ratio \( u/\bar{u} \) for all benchmarks for which we have an estimate of the number of solutions. The ratio \( u/\bar{u} \) had an average value of 0.981, with standard deviation of 0.052. Besides one benchmark (doublyLinkedList.sk.8.37, with value 0.41), all other benchmarks had \( u/\bar{u} > 0.87 \). In comparison, for SEARCHTREE_SAMPLER, the average was 0.996 and standard deviation 0.038. SEARCHTREE_SAMPLER also performed the worst on the benchmark doublyLinkedList.sk.8.37, with value 0.538, and all other benchmarks having \( u/\bar{u} > 0.92 \). UNI_GEN2 obtained an average of 1.000 and a standard deviation of 0.002, with a minimum value of 0.999. On doublyLinkedList.sk.8.37, UNI_GEN2 timed out, so we cannot compare on this benchmark.

We also present plots of the number of unique solutions produced over time, for two representative benchmarks. In Figure 5.5 we show the graph for benchmark s820a.7.4, where the number of samples produced is larger than the total number of solutions. We

\(^4\text{We could not perform power analysis to estimate the type II error rate because that would require a specific alternative hypothesis and we did not see any natural alternative hypothesis for the distribution of samples.}\)
see that the number of unique solutions grows very fast initially, and then stabilizes as we approach complete coverage of all solutions. **SearchTreeSampler** and **UniGen2**, on the other hand, produce solutions at a much slower rate. In Figure 5.6 we show benchmark enqueueSeqSK.sk.10.42, where the number of valid samples produced is much smaller than the total number of solutions. We can see that **QuickSampler** is able to generate unique solutions orders of magnitude faster than **SearchTreeSampler** and **UniGen2**. We also notice a distinctive step pattern in the graph. This happens because we produce the largest number of samples at the end of each sampling epoch, when the collection of known mutations is the largest.

In summary, we see that **SearchTreeSampler** and **UniGen2** are a bit closer to uniform sampling, but **QuickSampler** is still very close. In almost all cases the number of unique solutions generated was very close to the number that would be expected if the sampling was uniform, and we are able to produce new unique solutions at a faster rate than the other techniques.

### 5.2 SMTSampler Evaluation

We now present the evaluation of SMTSampler, focusing on the advantages of working over high-level SMT constraints directly, instead of converting them to SAT. We have implemented SMTSampler as an open source tool\(^5\) in C++. We used Z3 [24] as the constraint solver, which natively supports MAX-SMT queries. Each benchmark is run in one core, with a maximum virtual memory of 4 GB and a time budget of 1 hour. We stop each execution after generating 1 million solutions or reaching the timeout of 1 hour, whichever occurs first. The experiments were run on a machine with a 64-core Intel Xeon CPU E5-4640 and 264

\(^5\)The source code is available at [https://github.com/RafaelTupynamba/SMTSampler](https://github.com/RafaelTupynamba/SMTSampler).
GB of memory. For each epoch, we establish a maximum time budget of 20 minutes for the generation of all neighboring solutions. We also set a timeout of 5 seconds for each individual MAX-SMT solver call. If the call cannot be completed in 5 seconds, we remove the MAX-SMT soft constraints and retry the call with only the hard constraints (such as the condition to flip one bit in a bit-vector) for 5 more seconds. This allows us to still make progress and obtain some solutions in case the MAX-SMT problems are too expensive.

We compare two versions of SMTSampler. The first, abbreviated SMTbv, uses one soft constraint per bit-vector. The second, abbreviated SMTbit, uses soft constraints for each bit inside a bit-vector. As a baseline, we use a technique abbreviated SAT, which works by bit-blasting the SMT formula to convert it into a SAT formula and then sampling solutions from the SAT formula. This would be similar to applying the QUICKSAMPLER to the problem, although still taking advantage of our adaptive combination of mutations. The goal is to evaluate the advantage of operating directly over high-level SMT formulas, as opposed to bit-blasting. We did not compare against techniques such as the MCMC-based approach from Ambigen [43] because those are only applicable over constraints which are linear on each variable and would not be able to handle the general SMT-LIB benchmarks. Moreover, experimental evaluation of QUICKSAMPLER showed that QUICKSAMPLER is $1000 \times$ faster than MCMC-based approaches on SAT problems. SMTSampler focuses on enabling the sampling of solutions from complex SMT formulas, which are generally non-linear.

For the baseline bit-blasting approach, the expand_select_store rewriter option is used to replace $select(store(\ldots),\ldots)$ patterns by if-then-else terms. In addition, the Z3 tactics from Table 5.6 are applied to encode arrays as uninterpreted functions, apply Ackermann’s encoding to those functions, and bit-blast bit-vectors. In our experiments, we chose not to encode the SAT problem into conjunctive normal form (CNF) because we found that this conversion lead to slower solving due to the introduced auxiliary variables. Our conversion approach enables the conversion of most benchmarks into SAT, as long as they do not use the theory of arrays with extensionality, including equality comparisons between arrays.

**Coverage Metric**

When sampling from SMT formulas, we noticed that the number of unique solutions generated is an incomplete metric for coverage. Sometimes, it is easy to sample a large number

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simplify</td>
<td>Simplify the formula</td>
</tr>
<tr>
<td>bvarray2uf</td>
<td>Encode arrays as UFs</td>
</tr>
<tr>
<td>ackermannize_bv</td>
<td>Apply Ackermann’s encoding to UFs</td>
</tr>
<tr>
<td>bit-blast</td>
<td>Bit-blast the bit-vector variables</td>
</tr>
</tbody>
</table>
of solutions which are only trivially different and thus not interesting inputs for verification. For example, if a bit-vector variable $x$ of size 32 in a formula is only constrained by a condition such as $x > 5$, there are billions of values for $x$ that would satisfy this constraint. However, enumerating all those possibilities would probably not generate interesting inputs and a better strategy would be mutating other variables in the formula.

To better evaluate the coverage of the constraint space, we propose the use of a different coverage metric. We notice that the SMT formula has an abstract syntax tree (AST) structure where internal nodes better consolidate higher-level information than the leaf variable nodes. Thus, as a coverage metric we use coverage statistics about the internal nodes of the formula. For each internal node of type Boolean, we remember whether this node ever received the values of True or False in the generated solutions. Additionally, for internal nodes of type bit-vector, we remember if each of its bits ever received the values 1 or 0 in the generated solutions. The coverage metric is the number of such internal Booleans and bits which received both possible values among the set of generated solutions.

This metric can be thought of as a measure of the coverage of a circuit that evaluates the constraint. One could synthesize a circuit that takes as inputs assignments to the variables of the formula and produces a Boolean output of True or False indicating whether the formula is satisfied. The values computed by the internal nodes of the formula correspond to the intermediate values computed by the internal wires of this circuit. In this sense, the coverage metric we defined is equivalent to the coverage of internal wires in this circuit, when it is exercised by the generated solutions. Therefore, we use this metric as a proxy for the coverage that could be obtained when executing the design under test with the generated stimuli.

**Experimental Results**

Our benchmarks are obtained from SMT-LIB [7], specifically the problems in the logic QF_AUFBV and its sub-logics, such as QF_ABV, and QF_BV. The benchmarks include problems from the verification of hardware and software, bounded model checking, symbolic execution, static analysis and others.

We have tried all techniques on benchmarks from each directory available from those logics of SMT-LIB. Some directories had benchmarks which were inadequate for the problem, so we discarded them from the results. Those are cases where the formula is unsatisfiable, or the number of unique solutions that can be produced is less than 100, or where no coverage can be obtained. We ran the experiments over the remaining 22 directories, by randomly choosing 15 benchmarks from each, when there were at least 15 benchmarks available. A total of 274 benchmarks were chosen by following this procedure. From those, we excluded the benchmarks for which none of the techniques were able to produce more than one solution, leaving a final set of 213 benchmarks.

Table 5.7 shows the directories of benchmarks used, along with average statistics from the benchmarks in each directory. We first list the number $n$ of benchmarks which were used from each directory. All other values in the table are averages computed over those $n$
benchmarks in a directory. We list the number of internal nodes in the SMT formula, as a measure of the benchmark size. We also list the number of variables from each type Array, BV, Bool, UF. The 'bits' column represents the total number of bits in all the bit-vector and Boolean variables in the formula.

Then Table 5.8 presents average results from the experiments with the three techniques. First, we list the number of unique solutions produced, then the ratio of unique solutions over time and, finally, the total coverage obtained. Those values are also averages over the benchmarks in each directory. When computing the rate of unique solutions over time, we only include the time spent executing Z3 API calls. This is to ensure that the result is fair and not influenced by our implementation of the methods to store, process and combine solutions. In those Z3 API calls we include the time for solving constraints, checking the validity of solutions and converting solutions from SAT into SMT format. We do not include the time spent computing the coverage achieved by the solutions, as the coverage computation is done only for evaluation and is not required to apply the techniques.

Overall, we see that the SMT-based techniques tend to perform better than the bit-blasting approach. For a more thorough evaluation, we present graphs representing the rate of solution generation and the coverage on all 213 benchmarks.

Figure 5.7 compares the rate of generation of unique solutions between the SMT-based techniques and the SAT technique. The rates are defined as the number of unique solutions produced divided by the time spent in calls to the Z3 APIs. The y axis represents the logarithm in base 10 of these rates for both techniques. Higher bars indicate that the SMT-
Table 5.8: SMTSAMPLER results over the benchmarks.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Unique solutions</th>
<th>Unique solutions / second</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SMTbv</td>
<td>SMTbit</td>
<td>SAT</td>
</tr>
<tr>
<td>QF_AUFBV/ecc</td>
<td>209353</td>
<td>26860</td>
<td>49087</td>
</tr>
<tr>
<td>QF_ABV/bmc-arrays</td>
<td>807570</td>
<td>1229174</td>
<td>1096836</td>
</tr>
<tr>
<td>QF_ABV/stp_samples</td>
<td>258440</td>
<td>457702</td>
<td>287537</td>
</tr>
<tr>
<td>QF_ABV/dwp_formulas</td>
<td>895330</td>
<td>1303088</td>
<td>319558</td>
</tr>
<tr>
<td>QF_ABV/egt</td>
<td>769975</td>
<td>916283</td>
<td>1333783</td>
</tr>
<tr>
<td>QF_ABV/bench_ab</td>
<td>181716</td>
<td>341169</td>
<td>689368</td>
</tr>
<tr>
<td>QF_ABV/platania/_member</td>
<td>2085</td>
<td>113046</td>
<td>0</td>
</tr>
<tr>
<td>QF_BV/bmc-bv</td>
<td>439867</td>
<td>1250125</td>
<td>297913</td>
</tr>
<tr>
<td>QF_BV/bmc-bv-svcomp14</td>
<td>40899</td>
<td>5915</td>
<td>131089</td>
</tr>
<tr>
<td>QF_BV/spear/zebra_v0.95a</td>
<td>196822</td>
<td>18633</td>
<td>35</td>
</tr>
<tr>
<td>QF_BV/RWS</td>
<td>4776</td>
<td>476</td>
<td>201</td>
</tr>
<tr>
<td>QF_BV/gulwani-pld08</td>
<td>177745</td>
<td>153184</td>
<td>127608</td>
</tr>
<tr>
<td>QF_BV/stp_samples</td>
<td>505214</td>
<td>501507</td>
<td>438752</td>
</tr>
<tr>
<td>QF_BV/brummayerbiere2</td>
<td>263405</td>
<td>531815</td>
<td>712535</td>
</tr>
<tr>
<td>QF_BV/tacas07</td>
<td>33928</td>
<td>2444</td>
<td>28465</td>
</tr>
<tr>
<td>QF_BV/bench_ab</td>
<td>1109129</td>
<td>3385658</td>
<td>2783503</td>
</tr>
<tr>
<td>QF_BV/sage/app2</td>
<td>218827</td>
<td>213889</td>
<td>217598</td>
</tr>
<tr>
<td>QF_BV/sage/app9</td>
<td>1137172</td>
<td>1984923</td>
<td>1495072</td>
</tr>
<tr>
<td>QF_BV/sage/app8</td>
<td>389719</td>
<td>543033</td>
<td>260763</td>
</tr>
<tr>
<td>QF_BV/sage/app5</td>
<td>1146022</td>
<td>1397666</td>
<td>1008205</td>
</tr>
<tr>
<td>QF_BV/sage/app1</td>
<td>243452</td>
<td>509655</td>
<td>527177</td>
</tr>
<tr>
<td>QF_BV/sage/app12</td>
<td>470245</td>
<td>847513</td>
<td>334077</td>
</tr>
</tbody>
</table>

Figure 5.7: Speed comparison between the different approaches.

Based approach performed better than the SAT-based approach on that benchmark. For 23 benchmarks, the SAT approach was unable to produce any solutions because of a solver timeout. In these cases, the logarithm would be $+\infty$. Those are represented by bars that reach the top of the graph.

Figure 5.7 shows that, in general, the approaches that work over SMT formulas can generate more unique solutions in a given time budget, compared to bit-blasting. There were some benchmarks for which the SAT approach was able to generate more samples, such as some benchmarks from QF_ABV/egt and QF_ABV/bench_ab. Analyzing those benchmarks,
we found that they were mostly composed of Boolean operations from combinatorial logic, with very few bit-vector operations. It is natural that, in those cases, a SAT representation for the formula is more efficient and can be solved faster.

However, we also noticed that for many of those formulas, the larger number of solutions produced by SAT did not give any increase in the coverage metric. This reinforces our hypothesis that the speed to generate unique solutions is an incomplete metric, and we should also be analyzing the coverage obtained by the different approaches. Figure 5.8 presents the graphs comparing the coverage obtained by the SMT-based approaches and the SAT-based approach. Here, we see even more noticeable differences in favor of the SMT-based approaches, especially SMTbit. On the benchmarks from QF_ABV/bmc-arrays, QF_ABV/dwp_formulas, QF_BV/stp_samples and QF_BV/tacas07, for example, those approaches obtained significantly more coverage than SAT.

Overall, we see that the SMT-based approaches proposed by SMTSampler perform better than the baseline bit-blasting approach. They are more robust, being able to produce solutions and obtain coverage on a larger range of benchmarks, while also obtaining a higher constraint coverage and generating solutions at a higher speed in most cases. Between the two SMT-based approaches, we could not identify a clear winner. We noticed that the fine-grain soft constraints from SMTbit approach can help obtain more precise atomic mutations and produce higher coverage. However, SMTbv tends to be more robust on formulas where the MAX-SMT queries are harder to solve, since it uses a smaller number of soft constraints.

5.3 GuidedSampler Evaluation

We now present our evaluation of GuidedSampler, our coverage-guided sampler. We have implemented GUIDED_SAMPLER as an open-source tool\textsuperscript{6} in C++, using Z3 \cite{SUT_1} as the constraint solver. Z3 has native support for the MAX-SMT queries \cite{MAX_SMT} that are required by

\textsuperscript{6}The source code is available at https://github.com/RafaelTupynamba/GuidedSampler.
CHAPTER 5. EVALUATION

GuidedSampler. For evaluation, we have used 213 benchmarks from the QF_AUFBV logic of SMT-LIB [7], and its sub-logics, such as QF_ABV and QF_BV. The benchmarks come from 22 different families, which will be presented in Table 5.9. They are the same benchmarks which were used for the evaluation of SMTSampler, and include a diverse set of complex, non-linear constraints. We used a time budget of 30 minutes for each sampling experiment.

Coverage Predicates

We performed two sets of experiments, varying the way coverage predicates are generated. The goal is to evaluate the technique both using a general notion of coverage that stems from the formula itself and also a problem-specific notion of coverage, which can be quite different from the original constraint.

The first set of experiments uses internal predicates, which are subparts of the formula itself. Considering the representation of the formula $\phi$ as an abstract syntax tree (AST), we look at the internal nodes of type bit-vector or Boolean. For example, in the formula $\phi = \phi_1 \lor \phi_2$, two of the internal nodes are the disjuncts $\phi_1$ and $\phi_2$. We sample solutions to $\phi$ using a generic sampler, SMTSampler, and collect coverage statistics on those Boolean nodes and individual bits inside the bit-vector nodes. We identify a subset of those internal nodes that have exhibited diverse and independent coverage behaviors and choose the sub-formulas represented by those nodes as the coverage predicates. For example, in the formula $\phi = \phi_1 \lor \phi_2$, we could pick $\phi_1$ and $\phi_2$ to be two of the coverage predicates, and choose additional predicates as subparts of $\phi_1$ and $\phi_2$. This is intended to represent a generic coverage notion based on the formula itself. The number of predicates collected varies from only a couple on some benchmarks up to hundreds of predicates on others.

The second set of experiments uses random predicates, which are randomly generated constraints involving the variables in $\text{Vars}[\phi]$. The coverage predicates are randomly sampled from a context-free grammar that includes all arithmetical, logical and bit-wise operations from SMT-LIB, as well as the variables from $\text{Vars}[\phi]$. We generate from 16 to 48 predicates for each benchmark, with each predicate having on average a depth of 4 operations (i.e., any path from the root to a leaf in the formula AST has 4 operations on average). These random predicates are intended to represent a problem-specific measure of coverage, as they are independent of the original formula.

Approaches

In our evaluation, we compare the following seven approaches for coverage-guided sampling.

BC: Blocking coverage classes. This baseline approach consists of computing a solution $\sigma_0$ to $\phi$ and then adding a new blocking clause $(\psi_1 \neq \psi_1[\sigma_0] \lor \psi_2 \neq \psi_2[\sigma_0] \lor \cdots \lor \psi_n \neq \psi_n[\sigma_0])$ to the formula that guarantees that future solutions will not come from the coverage class $G[\sigma_0]$. We then compute a new solution $\sigma_1$ and add a new blocking clause that blocks
coverage class $G[\sigma_1]$, and so on. At most one solution can be obtained from each coverage class.

BH: Baseline with hard constraints. This simple baseline approach consists of choosing $n$ random Boolean values $b = (b_1, b_2, \ldots, b_n)$ that define a coverage class and trying to find one solution from that class. We then use the SMT solver to generate one solution to $\phi \land (\psi_1 = b_1) \land (\psi_2 = b_2) \land \cdots \land (\psi_n = b_n)$, where the predicates are added as hard constraints. Then repeat the process for different random vectors $b$. No solution is generated for a given $b$ if class $G_b$ is empty.

BS: Baseline with soft constraints. This baseline approach consists of choosing $n$ random Boolean values $b = (b_1, b_2, \ldots, b_n)$ that define a coverage class and trying to find the solution closest to that class. We use the query $\text{MAX-SMT}(\{\phi\}, S_b)$, where the predicates are added as soft constraints $S_b = \{\psi_1 = b_1, \psi_2 = b_2, \ldots, \psi_n = b_n\}$. Then repeat the process for different random vectors $b$. Each MAX-SMT call is guaranteed to eventually find a solution if $\phi$ is satisfiable, but the optimization problems can be expensive.

S0: SMTSampler. This baseline approach consists of applying SMTSampler to sample solutions to formula $\phi$. The coverage predicates are not used.

S1 = S0 + M1. This approach includes our modification M1, defined in Section 4.2, which consists of flipping the values of coverage predicates to generate neighboring solutions in function $\text{computeNeighboringSolutions}$. This ensures neighboring solutions come from neighboring coverage classes.

S2 = S0 + M1 + M2. This approach includes modifications M1 and also M2, which consists of discarding solutions from repeated coverage classes in lines 23 and 34 in Algorithm 3. A coverage class is considered repeated if we have already generated a solution from that class in the current epoch of the algorithm.

S3 = S0 + M1 + M2 + M3: GuidedSampler. This is the GUIDEDSAMPLER approach, including modifications M1, M2 and also M3, which randomizes the coverage class of the initial base solution in lines 3-7 of Algorithm 3.

Evaluating Diversity of Classes

We first evaluate the number of coverage classes reached by our sampling approaches. For most benchmarks, the total number of non-empty coverage classes is large, so a good coverage-guided sampler must find solutions from a large number of classes during the fixed time budget of 30 minutes and not keep visiting the same few classes multiple times.
Table 5.9: Average benchmark statistics for GuidedSampler.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>n</th>
<th>nodes</th>
<th>Array</th>
<th>BV</th>
<th>Bool</th>
<th>bits</th>
<th>preds</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUBV/ecc</td>
<td>4</td>
<td>179</td>
<td>0</td>
<td>27</td>
<td>7</td>
<td>1931</td>
<td>12</td>
</tr>
<tr>
<td>QF_ABV/bmc-arrays</td>
<td>3</td>
<td>855</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>53</td>
<td>82</td>
</tr>
<tr>
<td>QF_ABV/stp_samples</td>
<td>15</td>
<td>1139</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>192</td>
<td>58</td>
</tr>
<tr>
<td>QF_ABV/dwp_formulas</td>
<td>5</td>
<td>613</td>
<td>3</td>
<td>32</td>
<td>0</td>
<td>428</td>
<td>38</td>
</tr>
<tr>
<td>QF_ABV/egt</td>
<td>15</td>
<td>90</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>QF_ABV/bench_ab</td>
<td>15</td>
<td>314</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>QF_ABV/platania/member</td>
<td>12</td>
<td>595</td>
<td>10</td>
<td>89</td>
<td>0</td>
<td>2856</td>
<td>2</td>
</tr>
<tr>
<td>QF_BV/bmc-bv</td>
<td>10</td>
<td>256</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>134</td>
<td>6</td>
</tr>
<tr>
<td>QF_BV/bmc-bv-svcomp14</td>
<td>8</td>
<td>7518</td>
<td>0</td>
<td>205</td>
<td>1055</td>
<td>7607</td>
<td>223</td>
</tr>
<tr>
<td>QF_BV/spear/zebra_v0.95a</td>
<td>9</td>
<td>571</td>
<td>0</td>
<td>185</td>
<td>0</td>
<td>2012</td>
<td>21</td>
</tr>
<tr>
<td>QF_BV/RWS</td>
<td>9</td>
<td>1086</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>3628</td>
<td>95</td>
</tr>
<tr>
<td>QF_BV/gulwani-pldi08</td>
<td>5</td>
<td>1146</td>
<td>0</td>
<td>130</td>
<td>0</td>
<td>950</td>
<td>185</td>
</tr>
<tr>
<td>QF_BV/stp_samples</td>
<td>14</td>
<td>793</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>QF_BV/brummayerbie2</td>
<td>3</td>
<td>632</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>149</td>
<td>27</td>
</tr>
<tr>
<td>QF_BV/tacas07</td>
<td>3</td>
<td>8812</td>
<td>0</td>
<td>345</td>
<td>588</td>
<td>16620</td>
<td>91</td>
</tr>
<tr>
<td>QF_BV/bench_ab</td>
<td>13</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>QF_BV/sage/app2</td>
<td>13</td>
<td>231</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>206</td>
<td>7</td>
</tr>
<tr>
<td>QF_BV/sage/app9</td>
<td>8</td>
<td>271</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>391</td>
<td>17</td>
</tr>
<tr>
<td>QF_BV/sage/app8</td>
<td>15</td>
<td>978</td>
<td>0</td>
<td>93</td>
<td>0</td>
<td>1047</td>
<td>31</td>
</tr>
<tr>
<td>QF_BV/sage/app5</td>
<td>12</td>
<td>268</td>
<td>0</td>
<td>29</td>
<td>0</td>
<td>354</td>
<td>3</td>
</tr>
<tr>
<td>QF_BV/sage/app1</td>
<td>10</td>
<td>115</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>268</td>
<td>5</td>
</tr>
<tr>
<td>QF_BV/sage/app12</td>
<td>12</td>
<td>247</td>
<td>0</td>
<td>31</td>
<td>0</td>
<td>358</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.9 shows average statistics about the benchmarks. For each benchmark family, the column n lists the number of benchmarks used from that family. All the following columns present average numbers among all benchmarks in that family. First, we list the number of internal nodes in the SMT formula and the number of variables of type array, bit-vector and Boolean. The ‘bits’ column presents the total number of bits in the bit-vector and Boolean variables in the formula, and the ‘preds’ column represents the number of coverage predicates in the experiments that used internal predicates.

Table 5.10 shows experimental results using random predicates. We list, for each of the seven approaches, the total number of unique coverage classes found in the time budget. The numbers are also averages over the n benchmarks in each family. We only show the results for experiments with random predicates, but the results with internal predicates were similar. From Table 5.10, we can see that the GuidedSampler approach S3 was able to find the largest number of classes for most benchmarks. S2 tends to be the second best approach, with S1 being the third best one. This shows the effectiveness of our modifications M1, M2 and M3 in helping find more coverage classes. On only three benchmark families S0 or BC performed better. We found that this happened because the MAX-SMT queries were too complex for those benchmarks and were frequently timing out.

The following are average statistics of GuidedSampler for internal predicates (random predicates in parentheses). The percentage of time spent in Z3 API calls was 82% (88%). The percentage of candidate solutions that turned out to be real solutions to the formula was 76% (75%). The percentage of solutions which were unique (distinct solutions) was 65%
Table 5.10: Average number of unique coverage classes (using random predicates).

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>BC</th>
<th>BH</th>
<th>BS</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUFBV/ecc</td>
<td>1829</td>
<td>188</td>
<td>854</td>
<td>6579</td>
<td>826</td>
<td>969</td>
<td>11377</td>
</tr>
<tr>
<td>QF_ABFV/bmc-arrays</td>
<td>1263</td>
<td>44</td>
<td>660</td>
<td>1337</td>
<td>1903</td>
<td>19374</td>
<td>19122</td>
</tr>
<tr>
<td>QF_ABFV/stp_samples</td>
<td>39750</td>
<td>9831</td>
<td>15082</td>
<td>11360</td>
<td>9751</td>
<td>679380</td>
<td>744289</td>
</tr>
<tr>
<td>QF_ABFV/dwp_formulas</td>
<td>8162</td>
<td>7028</td>
<td>5688</td>
<td>404</td>
<td>2641</td>
<td>4227</td>
<td>31735</td>
</tr>
<tr>
<td>QF_ABFV/egt</td>
<td>16276</td>
<td>8656</td>
<td>30984</td>
<td>16432</td>
<td>15561</td>
<td>49864</td>
<td>480017</td>
</tr>
<tr>
<td>QF_ABFV/bench_ab</td>
<td>2254</td>
<td>151</td>
<td>3406</td>
<td>3118</td>
<td>2765</td>
<td>2530</td>
<td>6000</td>
</tr>
<tr>
<td>QF_ABFV/platania/no_init_multi_member</td>
<td>1344</td>
<td>15</td>
<td>399</td>
<td>83</td>
<td>84</td>
<td>235161</td>
<td></td>
</tr>
<tr>
<td>QF_BV/bmc-bv</td>
<td>724</td>
<td>19</td>
<td>122</td>
<td>437</td>
<td>201</td>
<td>200</td>
<td>208</td>
</tr>
<tr>
<td>QF_BV/bmc-bv-svcomp14</td>
<td>4352</td>
<td>178</td>
<td>2866</td>
<td>2004</td>
<td>16363</td>
<td>19503</td>
<td>23861</td>
</tr>
<tr>
<td>QF_BV/spear/zebra_v0.95a</td>
<td>3896</td>
<td>50</td>
<td>2797</td>
<td>618</td>
<td>8135</td>
<td>9622</td>
<td>7384</td>
</tr>
<tr>
<td>QF_BV/RWS</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>187</td>
<td>84</td>
<td>80</td>
<td>53</td>
</tr>
<tr>
<td>QF_BV/gulwani-pldi08</td>
<td>4991</td>
<td>130</td>
<td>2524</td>
<td>9605</td>
<td>499158</td>
<td>566758</td>
<td>622999</td>
</tr>
<tr>
<td>QF_BV/stp_samples</td>
<td>34669</td>
<td>18015</td>
<td>81052</td>
<td>5387</td>
<td>431305</td>
<td>908703</td>
<td>1068203</td>
</tr>
<tr>
<td>QF_BV/brummayerbierere2</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>QF_BV/tacas07</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>51</td>
<td>58</td>
<td>78</td>
</tr>
<tr>
<td>QF_BV/bench_ab</td>
<td>158</td>
<td>2</td>
<td>134</td>
<td>99</td>
<td>153</td>
<td>149</td>
<td>152</td>
</tr>
<tr>
<td>QF_BV/sage/app2</td>
<td>12639</td>
<td>19501</td>
<td>20581</td>
<td>1583</td>
<td>769464</td>
<td>1004453</td>
<td>1035877</td>
</tr>
<tr>
<td>QF_BV/sage/app9</td>
<td>4910</td>
<td>1940</td>
<td>8829</td>
<td>13025</td>
<td>180458</td>
<td>257994</td>
<td>318534</td>
</tr>
<tr>
<td>QF_BV/sage/app8</td>
<td>14451</td>
<td>1312</td>
<td>20627</td>
<td>14655</td>
<td>209256</td>
<td>269363</td>
<td>328543</td>
</tr>
<tr>
<td>QF_BV/sage/app5</td>
<td>15344</td>
<td>1577</td>
<td>23776</td>
<td>6428</td>
<td>300547</td>
<td>457180</td>
<td>628252</td>
</tr>
<tr>
<td>QF_BV/sage/app1</td>
<td>6307</td>
<td>346</td>
<td>5731</td>
<td>3458</td>
<td>19765</td>
<td>20768</td>
<td>21473</td>
</tr>
<tr>
<td>QF_BV/sage/app12</td>
<td>8650</td>
<td>4371</td>
<td>9065</td>
<td>11766</td>
<td>330629</td>
<td>608030</td>
<td>683599</td>
</tr>
</tbody>
</table>

(94%). Finally, the percentage of unique solutions which were from unique coverage classes (distinct classes) was 46% (54%).

Figure 5.9 shows a more thorough comparison of the number of coverage classes found across all benchmarks. We define $N_X$ as the number of unique coverage classes for which a solution was found using approach $X$ during our time budget. The graphs in Figure 5.9 show the ratio of classes found $N_{S3}/N_X$ between approach S3 and each of the baseline approaches $X$, in a logarithmic scale. Whenever no solutions were generated by one approach, the logarithm would evaluate the $+\infty$ or $-\infty$. This is represented on the graphs by bars that reach the top or the bottom of the graph.

Figure 5.9 includes results using both internal predicates and random predicates. From the graphs, we can see that the GUIDED_SAMPLER approach S3 is vastly superior to the baseline approaches on most benchmarks. Baselines BC, BH and BS tend to be expensive because they all require one new solver call for each new solution that is generated. In BC, the solver calls are also increasingly more expensive, as the formula grows larger with the addition of more blocking clauses. Baseline BH frequently tries to find solutions from empty classes, leading to unsatisfiable queries to the SMT solver. Baseline BS makes MAX-SMT queries which are satisfiable, but still expensive to solve. BS does not scale well because it still requires one MAX-SMT query for each new solution generated. The SMT_SAMPLER approach S0 mitigates this cost by using a small number of MAX-SMT queries to learn atomic mutations and then quickly combines those mutations to generate a large number of solutions. However, S0 ignores the coverage predicates, so it frequently keeps generating
CHAPTER 5. EVALUATION

Figure 5.9: Unique classes covered: GUIDED_SAMPLER vs. baselines.
Table 5.11: Mean ratios of the unique coverage classes reached $N_X$ and mean values of the uniformity metric $H_X$ for all approaches.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Internal Predicates</th>
<th>Random Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{S3}/N_{BC}$</td>
<td>mean $[low - high]$</td>
<td>mean $[low - high]$</td>
</tr>
<tr>
<td>$N_{S3}/N_{BH}$</td>
<td>3.4 $[0.45 - 26]$</td>
<td>4.2 $[0.44 - 40]$</td>
</tr>
<tr>
<td>$N_{S3}/N_{BS}$</td>
<td>2.6 $[0.23 - 28]$</td>
<td>38 $[1.5 - 1018]$</td>
</tr>
<tr>
<td>$N_{S3}/N_{S0}$</td>
<td>2.5 $[0.34 - 19]$</td>
<td>4.1 $[0.52 - 31]$</td>
</tr>
<tr>
<td>$N_{S3}/N_{S1}$</td>
<td>3.4 $[0.60 - 19]$</td>
<td>5.4 $[0.49 - 60]$</td>
</tr>
<tr>
<td>$N_{S3}/N_{S2}$</td>
<td>1.1 $[0.56 - 2.2]$</td>
<td>1.7 $[0.54 - 5.2]$</td>
</tr>
</tbody>
</table>

solutions from the same few classes.

Table 5.11 shows mean values for the ratios $N_{S3}/N_X$, in the format $10^\mu [10^{\mu-\sigma} - 10^{\mu+\sigma}]$, where $\mu$ and $\sigma$ are the average and standard deviation of log$_{10}(N_{S3}/N_X)$ across all benchmarks. The numbers show that S1, S2 and S3 can find a much larger number of classes when compared to S0. This demonstrates that M1 is the most important of our modifications. M1 ensures that the atomic mutations flip only a small number of the coverage predicates, so those mutations can be combined and generate solutions from more diverse classes. The modifications M2 and M3 also contribute to a small increase in the number of visited classes.

In Figure 5.10, we can see an example of the number of unique classes visited over time, for one representative benchmark. The GUIDEDSAMPLER approach S3 is the quickest to discover new classes, followed closely by approaches S2 and S1. We can also see that in this case the baseline approaches S0, BS, BH and BC can find a much smaller number of classes.

**Evaluating Uniformity over Coverage Classes**

In addition to looking at the number of classes covered, we also analyze the distribution of solutions among those classes. This is important, since a good coverage-guided sampler needs to sample from all the coverage classes with equal weight. Figure 5.11 shows the fraction of
Figure 5.10: Unique classes over time for one benchmark (with random predicates).

Figure 5.11: Uniformity of solutions over coverage classes for one benchmark (with random predicates).
solutions generated from each coverage class for the same benchmark of Figure 5.10. S0 is the most biased sampler, with the most frequent class being covered by 58% of the solutions. BC, BH and BS are very uniform, but could only reach less than 1100 classes. Our approach S3 is very close to uniform, with its line at the bottom of the graph, extending to the right until 115196 classes (not shown).

If $S$ solutions are generated, covering $N$ distinct classes, we expect each of those classes to be covered by approximately $S/N$ solutions. So as a first order estimate on how biased the distribution is, we look at the number $M$ of solutions found in the most frequent class. We define $H = M/(S/N)$ as a measure of how far the frequency $M$ is from the expected frequency $S/N$. The closer $H$ is to 1, the more uniform is the distribution of solutions found among those classes.

Table 5.11 displays mean values of $H$ across all benchmarks. Approach BC is guaranteed to have $H_{BC} = 1$, since each coverage class can only be sampled once. As expected, the baseline approaches BH and BS are also very uniform, since each new solution is generated inside or near a random coverage class. S0 is the most biased approach, often sampling a large number of solutions from a single class (in average, 14 or 19 times more often than it should). From the table, we see that our modifications M1 and M2 are essential for producing a more uniform distribution over the coverage classes, almost as uniform as the one from BH and BS, with $H_{S3} \approx 1.3$ for internal predicates and $H_{S3} \approx 2.9$ for random predicates. Those values show that in GuidedSampler the most frequent class is typically only oversampled by a factor of 3× or less.
Chapter 6

Related Work

This chapter discusses the related work in sampling from SAT and SMT constraints, as well as weighted sampling.

6.1 Sampling from SAT Constraints

There are several different techniques used to tackle the problem of sampling solutions to Boolean constraints [47]. The problem of sampling SAT witnesses is also closely related to the problem of counting the number of solutions, which has \#P-complete complexity. Several sampling techniques can be applied to model counting or use some form of model counting internally [70, 29, 48].

One class of sampling methods is based on Markov Chain Monte Carlo (MCMC) algorithms [42, 43]. These include simulated annealing and Metropolis-Hastings which are used to generate samples from a probability space. Those MCMC methods are guaranteed to eventually converge to the desired distribution (such as uniform sampling). However, this convergence is slow in practice for real-world problems, so the algorithms typically employ heuristics which make the sampling more biased [69, 42]. For example, [69] combines Metropolis steps with random walk steps through the assignments to the variables of the formula. In comparison, our techniques do not need to wait for a convergence time and try to cover the entire search space by finding solutions closest to randomly selected points.

One similar line of work attempts to modify the SAT solver search heuristics in order to generate a more diverse set of solutions [50]. However, this diverse sampling does not specify a desired distribution of solutions, such as uniform distribution, or the coverage of internal nodes of the formula, or a coverage-guided distribution. Our techniques do not modify the inner search strategies of SAT solvers, but instead uses the SAT solvers as an oracle to answer MAX-SAT queries.

SearchTreeSampler [29] seems to be the closest technique to QuickSampler in literature, by also using a SAT solver as an oracle. One difference is SearchTreeSampler performs simple satisfiability queries instead of the MAX-SAT queries by QuickSampler.
SearchTreeSampler works by exploring the tree of variable assignments in a breadth-first way, generating pseudosolutions, which are partial assignments to the variables that can be completed to a full solution. SearchTreeSampler uses a parameter \( k \) which specifies the number of samples computed per level in the tree and can be used to trade-off uniformity and number of solver calls required. On the other hand, QuickSampler uses a different strategy to cover the search space, and also generates a vastly larger number of samples per solver call, by combining learned mutations.

A different class of algorithms is based on universal hashing [28, 48] and can provide strong guarantees of uniformity. These techniques work by adding additional constraints to the formula (known as hash functions) in order to partition the search space uniformly. Those hash functions are typically formed by computing the XOR of a random subset of variables [33]. UniGen [18] and UniGen2 [15] are examples of this class, with the latter also employing parallelism to improve performance. In comparison, QuickSampler does not attempt to be perfectly uniform, but only close to uniform in practice. QuickSampler primarily aims for efficiency, using solver calls which are much less expensive to solve than the XOR constraints of hash functions, and generating a large number of samples per solver call.

### 6.2 Sampling from SMT constraints

As we have described, there is a large body of work in sampling solutions to Boolean satisfiability (SAT) formulas [47]. In principle, methods to sample solutions to SAT formulas can also be applied to SMT, as there are techniques for eager encoding of SMT formulas into SAT. However, one limitation of this conversion is the loss of the higher-level structure of the formula, which could be leveraged to generate samples more efficiently and also to ensure the samples are diverse. In SMTSampler, we have found that working at the SMT level without converting the formula into SAT leads to a larger number and diversity of samples.

One important class of sampling techniques is based on Markov Chain Monte Carlo (MCMC) methods [42, 43]. MCMC is the basis of most constrained-random verification techniques [43, 52, 73, 42]. MCMC techniques are typically effective for linear constraints, where the space of solutions is composed of polytopes which can be efficiently covered with random walks [42]. However, they are not so effective on arbitrary non-linear constraints, that lead to a more sparse distribution of solutions. SMTSampler and GuidedSampler, on the other hand, are designed to be applied to arbitrary, complex non-linear constraints.

A different strategy for sampling is using a constraint solver to produce each sample. The internal search heuristics of the solver can be modified to generate more diverse samples [50]. An important limitation of this approach is that it requires one constraint solver call per each sample produced, which is expensive. SMTSampler, on the other hand, generates several samples per solver call.

On the more theoretical side, techniques such as UniGen [18] and UniGen2 [15], based on universal hashing, can sample solutions from SAT formulas with a provably uniform
distribution. However, these techniques are expensive, as they require solving constraints which include complex hash functions that are hard to solve. In addition, the goal of sampling uniformly from the solution space does not necessarily lead to the best coverage of the constraint space for SMT constraints. We designed SMTSampler and GuidedSampler with the goal of efficiently generating solutions that cover well the constraint space of large and complex SMT formulas.

Following the universal hashing approach, SMTApproxMC [16] is an approximate model counter for SMT formulas. It is applicable only to formulas in the bit-vector theory and works similarly to UniGen [18] and UniGen2 [15], but using different hash functions that work at the word level. Although SMTApproxMC is a model counter, it could be adapted to work as a random sampler of bit-vector solutions, by outputting the solutions in a given cell, after the solution space is uniformly partitioned into cells. In contrast to SMTApproxMC, SMTSampler is designed to be more efficient and to also work over more general SMT formulas containing the theories of arrays, uninterpreted functions and bit-vectors.

6.3 Weighted Sampling

As discussed above, there is a large number of techniques dedicated to sampling solutions to logical constraints, especially for Boolean (SAT) constraints [47]. For example, techniques may be based on model counting [29], Markov Chain Monte Carlo (MCMC) [69, 43, 42], SAT solver search heuristics [50], combination of mutations [27, 26] and universal hashing [28, 48]. However, most sampling techniques have a general goal about the desired distribution of solutions, such as uniform distribution [18, 15], not allowing more specific notions of coverage.

Some of the above techniques, such as MCMC, can be adapted to the context of weighted sampling. However, they frequently fail to converge to the desired distribution in practice. There are techniques that can sample from a literal-weighted distribution [14], such as WAPS [34], however, not all distributions can be specified by assigning weights to literals in the formula. GuidedSampler, on the other hand, allows the desired distribution to be specified by coverage points, which are typically already present in testing and verification applications. GuidedSampler also enables sampling directly from SMT formulas, without a SAT conversion.
Chapter 7

Conclusion

In this dissertation, we presented three novel techniques for efficient sampling of solutions to logical constraints.

1. **QuickSampler**, a technique to sample solutions to SAT constraints, targeting a near uniform distribution.

2. **SMTSampler**, a technique to sample solutions to SMT constraints, targeting a good coverage of the constraint space defined by the SMT formula.

3. **GuidedSampler**, a technique for *coverage-guided sampling* of SMT solutions, where the distribution is guided by coverage points specified by the user.

Our techniques are publicly available as free and open-source tools, distributed under the BSD 3-Clause License. They allow the efficient generation of millions of solutions from only tens of queries to a constraint solver. They have been evaluated on large and complex benchmarks that come from real-world applications and have already started being used by other research groups [58, 57].

**QuickSampler**

QuickSampler is a new technique to sample solutions to Boolean constraints, with applications in constrained-random verification, symbolic execution and fuzz testing. By leveraging a small number of MAX-SAT solver calls, QuickSampler can generate millions of samples. QuickSampler works by computing some simple patterns of bit-flips, called *atomic mutations*, which can be applied to known solutions to generate another solution to the formula. It produces samples by combining $k$ such atomic mutations together, for each $k \leq 6$. Those samples are not guaranteed to be solutions for the formula, but they were solutions with high probability on hundreds of SAT benchmarks.

Our experiments show that the produced samples are valid with an average probability of 75% on a set of large, real-world benchmarks. Moreover, QuickSampler is more than 2
orders of magnitude faster at producing valid samples, when compared to other state-of-the-art samplers. It is also more than 2 orders of magnitude faster at producing unique valid samples, which is specially important to increase testing coverage. We have also verified that QuickSampler is still 1 order of magnitude faster even when it takes the additional time to verify that the generated solutions are valid. Finally, the distribution of samples produced is close to uniform on most of the benchmarks.

**SMTSampler**

SMTSampler is a new technique for efficient stimulus generation from SMT constraints. It has a goal of generating solutions that maximize the internal coverage of the constraint. SMTSampler leverages the same idea of computing atomic mutations and combining them to generate samples. However, SMTSampler is adapted to work over the higher-level theories of bit-vectors, arrays, and uninterpreted functions, producing and combining mutations over those data types directly. This leads to more efficient solving, while also eliminating the cost to convert the SMT formula into SAT. We show in our experimental evaluation that on most benchmark programs SMTSampler outperforms a naïve approach that converts an SMT formula to SAT and then applies QuickSampler. Moreover, unlike QuickSampler, SMTSampler only outputs valid samples and adaptively increases the number $k$ of atomic mutations combined based on the accuracy in the samples that are tried. Our evaluation over a large set of industrial SMT benchmarks shows that working over SMT solutions allows SMTSampler to be effective on a larger set of formulas, generate more unique samples and obtain a better coverage of the constraint space.

**GuidedSampler**

Finally, we introduced the problem of coverage-guided sampling, to allow shaping the distribution of SMT solutions via user-specified coverage predicates. GuidedSampler is a novel technique developed for coverage-guided sampling of SMT solutions. GuidedSampler extends SMTSampler with 3 important modifications to allow coverage-guided sampling based on arbitrary coverage predicates. Our modifications use information about the coverage class of the solutions in order to guide the search into exploring new classes of solutions and avoid sampling from the same repeated classes. Our experimental evaluation shows that GuidedSampler is able to reach 2.5× to 38× more coverage classes than the baseline approaches, while having a distribution over coverage classes that is close to uniform. The approach works well for both internal coverage predicates based on the formula itself and also for random coverage predicates, showing a good potential for applicability in a diverse range of applications. Even for applications where a general notion of coverage is suitable, our evaluation shows that GuidedSampler can outperform SMTSampler in achieving this coverage.
7.1 Future Work

Our novel idea of computing atomic mutations and combining them to efficiently generate new samples has shown a great promise as a practical technique for constraint sampling. It is able to efficiently generate millions of diverse solutions to large and complex constraints, and can also be adapted to different target distributions. We now list some interesting future research directions, including new applications, a better understanding of the underlying structure and guarantees, new ideas for improved performance, and the exploration of coverage-guided sampling.

Applications

One important research direction is in evaluating different applications for the sampling techniques. Our techniques could be used to find bugs and security vulnerabilities in both hardware and software. They could be combined with existing testing and verification techniques for improved efficiency and scalability. And they could even be applied to new domains, such as synthesis problems. We present below some research ideas in different applications.

Symbolic Execution and Fuzzing. Conventional symbolic execution [41, 21] and dynamic symbolic execution techniques [31, 63, 13, 12, 20, 46, 68, 3, 56, 2, 40, 61, 4, 59, 62, 32, 66, 5] are well-established techniques used in the testing of software, but which have also been applied to the testing of firmware [19] and even hardware designs written in the Chisel hardware description language [6]. They work by computing a path constraint for each prefix of feasible execution paths in a program and using an SMT solver to generate a solution for each such constraint. However, in practice, these techniques face scalability problems because the number of paths for any reasonable program is astronomically large. Instead of generating a single solution for the path constraint of a path prefix, one could generate multiple solutions with SMTSampler to randomly test multiple paths having the same prefix. We call this approach constraint-based fuzzing. If multiple solutions could be generated efficiently, this would significantly speedup symbolic execution and reap the benefits of random testing [71, 72, 10, 36, 37, 54, 30]. This approach can be applied both to software and hardware designs. We have previously implemented one prototype symbolic execution engine for Chisel designs, by leveraging an interpreter for the FIRRTL intermediate language [39]. However, our symbolic execution engine faced scalability problems on large, complex circuits. We hope that, by using the SMTSampler approach, we will be able to better leverage the SMT solvers, obtaining many solutions that can increase coverage with a small number of solver calls. This was, in fact, the initial motivation for the design of our sampling techniques. We also plan to apply this idea to software testing, in a new approach combining the KLEE [13] symbolic execution engine and AFL [72] fuzz testing tool.

Constrained-Random Verification (CRV). For hardware testing, constrained-random verification (CRV) [52] is heavily used in industry to generate high-quality inputs for hard-
ware designs. In CRV, verification engineers specify preconditions required by the hardware and other constraints based on domain-specific knowledge [73, 51]. Multiple random inputs satisfying the constraints are then generated using a constraint solver that can sample random solutions from a constraint. For this approach, we could leverage high-level designs written in Chisel. One idea is to formally specify the interfaces, such as TileLink [22], used to connect different modules in a circuit. After those constraints are specified as SMT formulas, we can use the SMTSampler technique to generate a large number of valid inputs that exercise a given module. Another future direction is in adapting our sampling technique to be applied to SystemVerilog constraints, since SystemVerilog is the most common format used in industry for logical constraints in hardware circuits.

Other SMT Theories. One additional extension is supporting other SMT theories besides bit-vectors, arrays and uninterpreted functions. We could identify which additional theories are important for practical applications and see if our mutation combination function can be adapted to work over variables from the new theories.

Counterexample-Guided Inductive Synthesis (CEGIS). CEGIS [64] is an approach for synthesis consisting of a learner algorithm which generates candidate expressions and a verifier which checks them for correctness and produces counterexamples for invalid candidates. The counterexamples are then used as feedback for the learner to produce new candidates. This CEGIS loop is used, for example, in the enumerative approach to Syntax-Guided Synthesis (SyGuS) [1], the problem of synthesizing an expression constrained by a formal syntactic grammar. In a typical CEGIS loop, the verifier uses a constraint solver to generate one counterexample for each invalid candidate it receives. However, the counterexample may not be general enough to exclude a large class of invalid expressions, which will lead to the repetition of several loop iterations. We believe our sampling technique could be a good enhancement to CEGIS. By generating several diverse counterexamples, the verifier can provide more information to the learner so that it can make more progress on its own, limiting the number of calls to the verifier.

Understanding and Guarantees

Another important direction of research is in getting a deeper understanding of the combination of mutations. We have some intuitive understanding as to why our algorithms generate valid samples with high probability, but more work should be done into identifying and measuring the root causes for this accuracy. One direction we plan to pursue is in statistically analyzing the structure of the clauses that are present in our benchmarks and dependencies between different variables. We could obtain distributions for the number of bits that are flipped by each atomic mutation, the intersection of bits flipped by different mutations, the number of clauses affected by each of these bits and their structure. It is also important to look at the relationships between variables inside and outside the independent
support of the formula for our Boolean benchmarks. In addition, for GuidedSampler we should make further measurements to explain why the coverage predicates tend to follow the same mutation pattern as the regular Boolean and bit-vector variables of the formula when applying our mutation strategy.

Another important goal is to characterize the structure of the benchmarks for which our mutation strategy provides high accuracy. We have surprisingly found that our techniques work well over a very diverse range of benchmarks coming from several different domains in SMT-LIB. However, we believe our mutation combination would not have a high accuracy over randomly generated benchmarks. So there should be some common structure that is present in benchmarks from practical applications that makes them susceptible to the application of our sampling techniques. Understanding this structure could lead into further insights as to what problems would be good applications for the techniques.

These efforts in better understanding the techniques might also provide stronger guarantees on the effectiveness of the techniques. We could search for statistical or theoretical guarantees on the probability of producing valid samples and the distribution of samples that is generated. Such guarantees would provide a greater confidence when applying the techniques to current and new domains.

Algorithmic Improvements

Future research can also look into new ways to improve the efficiency and scalability of our sampling techniques. We list some interesting ideas of algorithmic improvements below.

Solver Internals. One idea in this space is to modify the internal search heuristics of SAT and SMT solvers, instead of treating the solvers as a black-box. We could study, for example, if different SAT polarity choices could improve our sampling algorithms. One could also apply different solving strategies or use other existing solvers for efficiency. For example, the Boolector [11] solver is reported to be more efficient than Z3 [24] in dealing with bit-vector constraints, so it might be a better choice for a large class of benchmarks. A better study of how lazy and eager approaches interplay with SMTSampler would also be interesting.

MAX-SAT Algorithm. The MAX-SAT optimizing solver is where our techniques spend most of their time. Therefore, an improvement to the MAX-SAT algorithm could lead to a significant speedup in the generation of samples. One simple direction is to try out different MAX-SAT algorithms and check which one performs better for our problems. One particularly interesting approach is to use a MAX-SAT algorithm which is anytime [49], meaning that it can be stopped at any point in time and will output the best solution found so far. This is a nice approach for dealing with MAX-SAT and MAX-SMT problems which are too large to solve, since we could still obtain a good quality solution in our time budget and make progress, even if it may not be the optimal solution. Along the same line, we
could also study whether this optimality is actually required, or whether our combination of mutations would still obtain a high accuracy when the neighboring solutions are not the closest ones to the base solution.

**Efficient SMT Evaluator.** Another algorithmic improvement would be in utilizing a technique such as SMT-JIT [44] for efficient evaluation of SMT formulas. SMT-JIT is a just-in-time compiler which can compile formulas from the QF_AUFBV logic of SMT into efficient LLVM [45] and machine code. SMTSAMPLER and GUIDEDSAMPLER could leverage SMT-JIT in the checking phase to very efficiently check if a candidate solution actually satisfies the formula and to compute its coverage class. In addition, a very significant speedup might be obtained if we can leverage the efficient SMT evaluator in the MAX-SAT algorithm. This might require making some adaptations to the current MAX-SAT algorithm.

**Coverage-guided Sampling**

Finally, one important direction is in further exploration of the problem of *coverage-guided sampling*. In our technique GUIDEDSAMPLER, we assumed that coverage predicates are provided by the user. We then focused on the problem of sampling from different coverage classes with equal weight. However, a crucial problem is studying how to choose coverage predicates that lead to a good ‘quality’ in the partitioning of the solution space. We devised one such strategy, *internal predicates*, based on our prior experience with SMTSAMPLER, in attempting to obtain a good coverage of the formula itself. But the exploration of more strategies to create coverage predicates still merits more research, and it might require some more application-specific experience.
Bibliography


Appendix A

Academic Genealogy

Thanks to the Mathematics Genealogy Project\(^1\), Wikipedia\(^2\) and some help from Rohan Padhye\(^3\), I reconstructed these two lines from my academic genealogy. The Mathematics Genealogy Project is a service of North Dakota State University and the American Mathematical Society. Full genealogical tree available at https://people.eecs.berkeley.edu/~rtd/genealogy.pdf.

I was honored to find that my academic genealogical tree includes giants of mathematics and physics Gauss, Isaac Newton, Kepler, Galileo, Tycho Brahe and Copernicus. It includes pioneers in concolic testing (Koushik Sen), the actor model of computation (Carl Hewitt and Gul Agha), genetic algorithms (John Henry Holland\(^4\)), constructionism and artificial intelligence (Papert), the first Turing-complete electronic computer (Arthur Burks), population genetics (G. H. Hardy), experimental psychology (Wundt and Edward B. Titchener), control systems theory (Edward Routh), matrix multiplication and abstract groups (Arthur Cayley), short-period comets (Johann Franz Encke), modern geology (Adam Sedgwick\(^5\)), galvanometers (Johann Schweigger), planet Uranus (Johann Elert Bode), electrostatic printing (Georg Christoph Lichtenberg\(^6\)), reactive water turbines (Johann Andreas Segner), numerical integration (Roger Cotes), infinitesimal calculus (Isaac Barrow), pressure measurement (Evangelista Torricelli), acoustics (Mersenne), refraction (Willebrord Snellius), modern embryology (Hieronymus Fabricius), condoms (Falloppio), modern human anatomy (Vesalius and Realdo Colombo), astronomical rings (Gemma Frisius), the Gymnasium system of secondary education (Johannes Sturm), solving cubic equations (Niccolò Fontana Tartaglia), accounting (Pacioli\(^7\)), scientific printing press (Regiomontanus), harmonic series (Oresme), trigonometry (Nasir al-Din al-Tusi), and algebraic geometry (Sharaf al-Dın al-Ṫusı).

\(^1\)https://www.genealogy.math.ndsu.nodak.edu/.
\(^2\)https://www.wikipedia.org/.
\(^3\)https://people.eecs.berkeley.edu/~rohanpadhye/.
\(^4\)One of the first people in the world to receive a Ph.D. in what might be called a “Computer Science” program in 1959 (officially “Communication Sciences Program” and “Logic of Computers Group”).
\(^5\)Also Charles Darwin’s advisor.
\(^6\)Also creator of A4 paper size system (unfortunately not adopted by this dissertation).
\(^7\)Also a teacher of Leonardo da Vinci.
Additionally, the genealogical tree also includes Christian humanist philosopher Erasmus, some prominent Lutheran reformers (Melanchthon and others), a leader and reformer of the Church of England (Thomas Cranmer), an Eastern Orthodox saint (Gregory Palamas), a Roman Catholic cardinal bishop (Bessarion), and two chief ministers to the Byzantine emperor (Demetrios Kydones and Theodore Metochites), as well as Leonardo da Vinci’s teacher John Argyropoulos, Gottfried Leibniz’s father (Friedrich Leibniz), and Charles Darwin’s son (George Darwin).
<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rafael Tupynambá Dutra</td>
<td>University of California, Berkeley</td>
<td>2019</td>
</tr>
<tr>
<td>Koushik Sen</td>
<td>University of Illinois at Urbana-Champaign</td>
<td>2006</td>
</tr>
<tr>
<td>Gul Agha</td>
<td>University of Michigan</td>
<td>1985</td>
</tr>
<tr>
<td>John Henry Holland</td>
<td>University of Michigan</td>
<td>1959</td>
</tr>
<tr>
<td>Arthur Burks</td>
<td>University of Michigan</td>
<td>1941</td>
</tr>
<tr>
<td>Cooper Harold Langford</td>
<td>Harvard University</td>
<td>1924</td>
</tr>
<tr>
<td>Edwin Boring</td>
<td>Cornell University</td>
<td>1914</td>
</tr>
<tr>
<td>Edward B. Titchener</td>
<td>Universität Leipzig</td>
<td>1892</td>
</tr>
<tr>
<td>Eilhard Wiedemann</td>
<td>Universität Leipzig</td>
<td>1872</td>
</tr>
<tr>
<td>Karl Christian Bruhns</td>
<td>Friedrich-Wilhelms-Universität zu Berlin</td>
<td>1856</td>
</tr>
<tr>
<td>Johann Franz Encke</td>
<td>Universität Göttingen</td>
<td>1825</td>
</tr>
<tr>
<td>Carl Friedrich Gauss</td>
<td>Universität Helmsted</td>
<td>1799</td>
</tr>
<tr>
<td>Johann Friedrich Pfaff</td>
<td>Universität Göttingen</td>
<td>1786</td>
</tr>
<tr>
<td>Abraham Gottelf Pfaff</td>
<td>Universität Leipzig</td>
<td>1739</td>
</tr>
<tr>
<td>Christian Augustus Planer</td>
<td>Universität Wittenberg</td>
<td>1686</td>
</tr>
<tr>
<td>Georg Pasch</td>
<td>Universität Wittenberg</td>
<td>1683</td>
</tr>
<tr>
<td>Michael Walther der Jüngere</td>
<td>Universität Wittenberg</td>
<td>1687</td>
</tr>
<tr>
<td>Johannes Andreas Quenstedt</td>
<td>Universität Wittenberg</td>
<td>1644</td>
</tr>
<tr>
<td>Christoph Notnagel</td>
<td>Universität Wittenberg</td>
<td>1630</td>
</tr>
<tr>
<td>Ambrosius Rhode</td>
<td>Universität Wittenberg</td>
<td>1610</td>
</tr>
<tr>
<td>Johannes Kepler</td>
<td>Tübingen Stift</td>
<td>1591</td>
</tr>
<tr>
<td>Tycho Brahe</td>
<td>Universität Wittenberg</td>
<td>1565</td>
</tr>
<tr>
<td>Caspar Peucer</td>
<td>Universität Wittenberg</td>
<td>1545</td>
</tr>
<tr>
<td>Georg Joachim Rheticus</td>
<td>Universität Wittenberg</td>
<td>1535</td>
</tr>
<tr>
<td>Nicolaus Copernicus</td>
<td>Universität di Bologna</td>
<td>1499</td>
</tr>
<tr>
<td>Domenico Maria Novara da Ferrara</td>
<td>Universität degli Studi di Firenze</td>
<td>1483</td>
</tr>
<tr>
<td>Regionomontanus</td>
<td>Universität Wien</td>
<td>1457</td>
</tr>
<tr>
<td>Basilius Bessarion</td>
<td>Mystras</td>
<td>1436</td>
</tr>
<tr>
<td>Gemistus Pletio</td>
<td>Mystras</td>
<td>1393</td>
</tr>
<tr>
<td>Demetrios Kydones</td>
<td>Thessaloniki</td>
<td>c.1365</td>
</tr>
<tr>
<td>Neilos Kabasilas</td>
<td>Thessaloniki</td>
<td>1363</td>
</tr>
<tr>
<td>Gregory Palamas</td>
<td>Constantinople</td>
<td>c.1336</td>
</tr>
<tr>
<td>Theodore Metochites</td>
<td>Constantinople</td>
<td>1315</td>
</tr>
<tr>
<td>Manuel Bryennios</td>
<td>Constantinople</td>
<td>c.1300</td>
</tr>
<tr>
<td>Gregory Choniades</td>
<td>Ilkhans Court at Tabriz</td>
<td>1296</td>
</tr>
<tr>
<td>Shams al-Din al-Bukhārī</td>
<td>Maragheh Observatory</td>
<td>c.1260</td>
</tr>
<tr>
<td>Nasir al-Din al-Tusi</td>
<td>Maragheh Observatory</td>
<td>c.1230</td>
</tr>
<tr>
<td>Kamal al-Din ibn Yunus</td>
<td>Persia</td>
<td>c.1200</td>
</tr>
<tr>
<td>Sharaf al-Din al-Tūsī</td>
<td>Persia</td>
<td>c.1175</td>
</tr>
</tbody>
</table>
APPENDIX A. ACADEMIC GENEALOGY

Rafael Tupynambá Dutra
Computer Scientist, Mathematician, Engineer

Koushik Sen
Computer Scientist

Gul Agha
Computer Scientist

John Henry Holland
Psychologist, Electrical Engineer and Computer Scientist

Arthur Burks
Mathematician, Physicist, Philosopher and Computer Scientist

Cooper Harold Langford
Analytic Philosopher and Mathematical Logician

Edwin Boring
Experimental Psychologist and Historian of Psychology

Edward B. Titchener
Psychologist

Eilhard Wiedemann
Physicist and Historian of Science

Karl Christian Bruhns
Astronomer

Johann Franz Encke
Astronomer

Carl Friedrich Gauss
Mathematician and Physicist

Johann Friedrich Pfaff
Mathematician and Astronomer

Abraham Gotthelf Kästner
Mathematician, Epigrammatist, Philosopher, Logician, Jurist and Physicist

Christian August Hausen
Mathematician and Physicist

Johann Andreas Planer
Mathematician and Philosopher

Georg Pasch
Logician and Protestant Theologian

Michael Walther der Jüngere
Mathematician and Lutheran Theologian

Johannes Andreas Quenstedt
Lutheran Dogmatician, Geographer, Philosopher, Logician and Theologian

Christoph Notnagel
Mathematician and Astronomer

Ambrosius Rhode
Mathematician, Astronomer and Physician

Johannes Kepler
Astronomer, Mathematician, Astrologer and Physicist

Tycho Brahe
Nobleman, Astronomer, Astrologer, Alchemist and Writer

Caspar Peucer
Reformer, Physician, Mathematician and Astronomer

Georg Joachim Rheticus
Mathematician, Astronomer, Cartographer, Navigational-instrument Maker, Medical Practitioner and Teacher

Nicolaus Copernicus
Polymath, Mathematician, Astronomer, Physician, Classics Scholar, Translator, Governor, Diplomat, Economist and Canon Lawyer

Domenico Maria Novara da Ferrara
Mathematician and Astronomer

Regiomontanus
Mathematician and Astronomer

Basilius Bessarion
Roman Catholic Cardinal Bishop

Gemistus Pletho
Philosopher

Demetrios Kydones
Theologian, Translator, Writer and Influential Statesman

Neilos Kabasilas
Palamite Theologian

Gregory Palamas
Theologian and Ecclesiastical Figure

Theodore Metochites
Statesman, Author, Gentleman Philosopher and Patron of the Arts

Manuel Bryennios
Astronomer, Mathematician and Musical Theorist

Gregory Choniades
Astronomer

Shams al-Dīn al-Bukhārī
Mathematician and Astronomer

Nasir al-Dīn al-Tusi
Polymath, Architect, Philosopher, Physician, Scientist and Theologian

Kamal al-Dīn ibn Yunus
Mathematician, Chemist and Jurist

Sharaf al-Dīn al-Tūsī
Mathematician and Astronomer
<table>
<thead>
<tr>
<th>Name</th>
<th>University</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rafael Tupynambá Dutra</td>
<td>University of California, Berkeley</td>
<td>2019</td>
</tr>
<tr>
<td>Koushik Sen</td>
<td>University of Illinois at Urbana-Champaign</td>
<td>2006</td>
</tr>
<tr>
<td>Gul Agha</td>
<td>University of Michigan</td>
<td>1985</td>
</tr>
<tr>
<td>Carl Hewitt</td>
<td>Massachusetts Institute of Technology</td>
<td>1971</td>
</tr>
<tr>
<td>Seymour Papert</td>
<td>University of Cambridge</td>
<td>1959</td>
</tr>
<tr>
<td>Frank Smithies</td>
<td>University of Cambridge</td>
<td>1937</td>
</tr>
<tr>
<td>G. H. Hardy</td>
<td>University of Cambridge</td>
<td>1903</td>
</tr>
<tr>
<td>E. T. Whittaker</td>
<td>University of Cambridge</td>
<td>1895</td>
</tr>
<tr>
<td>George Darwin</td>
<td>University of Cambridge</td>
<td>1871</td>
</tr>
<tr>
<td>Edward Routh</td>
<td>University of Cambridge</td>
<td>1857</td>
</tr>
<tr>
<td>Isaac Todhunter</td>
<td>University of Cambridge</td>
<td>1848</td>
</tr>
<tr>
<td>William Hopkins</td>
<td>University of Cambridge</td>
<td>1830</td>
</tr>
<tr>
<td>Adam Sedgwick</td>
<td>University of Cambridge</td>
<td>1811</td>
</tr>
<tr>
<td>Thomas Jones</td>
<td>University of Cambridge</td>
<td>1782</td>
</tr>
<tr>
<td>Thomas Postlethwaite</td>
<td>University of Cambridge</td>
<td>1756</td>
</tr>
<tr>
<td>Stephen Whisson</td>
<td>University of Cambridge</td>
<td>1742</td>
</tr>
<tr>
<td>Walter Taylor</td>
<td>University of Cambridge</td>
<td>1723</td>
</tr>
<tr>
<td>Robert Smith</td>
<td>University of Cambridge</td>
<td>1715</td>
</tr>
<tr>
<td>Roger Cotes</td>
<td>University of Cambridge</td>
<td>1706</td>
</tr>
<tr>
<td>Isaac Newton</td>
<td>University of Cambridge</td>
<td>1668</td>
</tr>
<tr>
<td>Isaac Barrow</td>
<td>University of Cambridge</td>
<td>1652</td>
</tr>
<tr>
<td>Vincenzo Viviani</td>
<td>Università di Pisa</td>
<td>1642</td>
</tr>
<tr>
<td>Evangelista Torricelli</td>
<td>Università degli Studi di Roma La Sapienza</td>
<td>c.1641</td>
</tr>
<tr>
<td>Benedetto Castelli</td>
<td>Università degli Studi di Padova</td>
<td>1610</td>
</tr>
<tr>
<td>Galileo Galilei</td>
<td>Università di Pisa</td>
<td>1585</td>
</tr>
<tr>
<td>Ostilio Ricci</td>
<td>Università di Brescia</td>
<td>c.1570</td>
</tr>
<tr>
<td>Niccolò Fontana Tartaglia</td>
<td>Università degli Studi di Padova</td>
<td>c.1516</td>
</tr>
</tbody>
</table>
Rafael Tupynambá Dutra  Computer Scientist, Mathematician, Engineer
Koushik Sen  Computer Scientist
Gul Agha  Computer Scientist
Carl Hewitt  Computer Scientist, Mathematician and Logician
Seymour Papert  Mathematician, Computer Scientist, Educator and Cognitive Scientist
Frank Smithies  Mathematician
G. H. Hardy  Mathematician
E. T. Whittaker  Mathematician and Mathematical Physicist
George Darwin  Barrister, Astronomer and Mathematician
Edward Routh  Mathematician and Mathematical Physicist
Isaac Todhunter  Mathematician
William Hopkins  Mathematician and Geologist
Adam Sedgwick  Geologist and Priest
Thomas Jones  Mathematician
Thomas Postlethwaite  Clergyman and Mathematician
Stephen Whisson  Mathematician
Walter Taylor  Mathematician and Classicist
Robert Smith  Mathematician
Roger Cotes  Mathematician
Isaac Newton  Natural Philosopher, Mathematician, Physicist, Astronomer, Theologian, Author, Economist, Alchemist and Biblical Chronologist
Isaac Barrow  Christian Theologian, Mathematician and Physicist
Vincenzo Viviani  Mathematician and Physicist
Evangelista Torricelli  Physicist and Mathematician
Benedetto Castelli  Mathematician
Galileo Galilei  Polymath, Astronomer, Physicist, Engineer, Natural Philosopher and Mathematician
Ostilio Ricci  Mathematician
Niccolò Fontana Tartaglia  Mathematician, Engineer, Topographic Surveyor and Bookkeeper
Appendix B

Author’s Biography

Rafael has participated in math olympiads since 12 years old, having obtained 22 medals, including 2 medals at the International Mathematical Olympiad (IMO). He has also received 3 gold medals at the Brazilian Physics Olympiad. Rafael got his Bachelor in Automation and Control Engineering from UFMG (2014), where he was admitted in 2009 with the highest score among all 63249 applicants. Due to his experience with math olympiads, he had the opportunity to concurrently obtain a Master in Mathematics degree from UFMG (2014). He also studied abroad at the University of Melbourne for one year (2012-2013). Rafael is completing his PhD in Computer Science at UC Berkeley (2019) advised by Koushik Sen, in the major area of Programming Systems, minor area of Security and outside minor of Mathematical Logic. After failing his first attempt at the preliminary examination in Programming Systems for the PhD, he apparently studied hard enough for the second attempt to be awarded the Tong Leong Lim Pre-Doctoral Prize for outstanding performance. In 2017, he participated as a coordinator of the International Mathematical Olympiad in Brazil and was lucky to have the joy of solving Problem 3, the hardest problem in the history of the IMO at the time. Rafael has one year of teaching experience, including one term as an acting instructor for Computer Security. He has not held a real job so far, but completed internships at Top Free Games, Google and Intel.