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

Fall 2019

Efficient Sampling of SAT and SMT Solutions for Testing and Verification

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%),

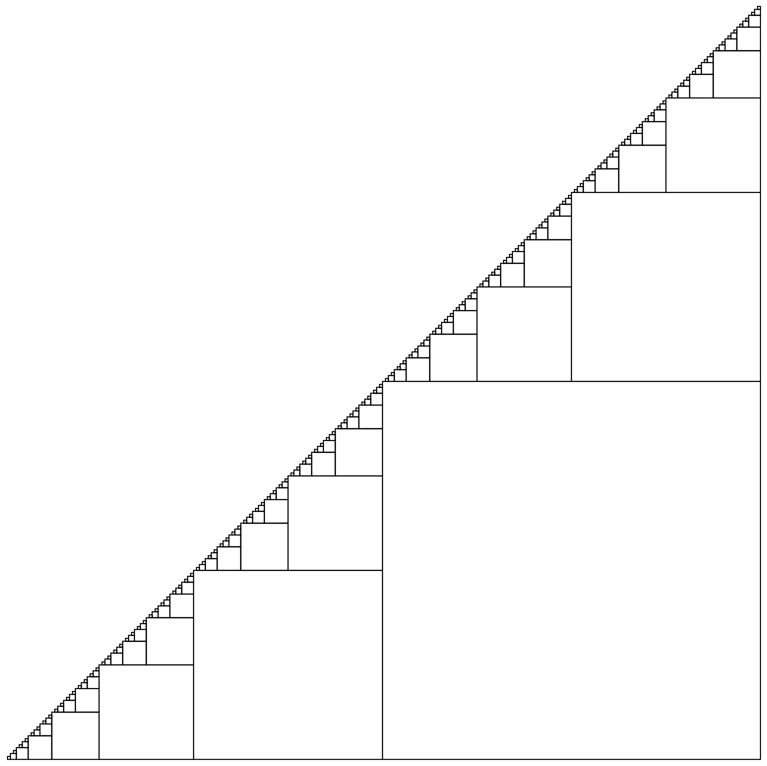
and the invalid samples can be easily filtered out in a post-processing step. Our evaluation of QuickSampler on large real-world benchmarks shows that it can produce unique valid solutions orders of magnitude faster than other state-of-the-art sampling tools. We have also empirically verified that the 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.



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

List of Figures

List of Tables

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, and my uncle Guilherme for first introducing me to programming in C and GNU/Linux. I especially want to thank my boyfriend and friends I met in Berkeley for making my life here so special, and the volleyball team at Cal for helping me live healthier and happier.

For the insightful discussions and feedback, I 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. 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 Boolean satisfiability (SAT) [49] constraints, satisfiability modulo theories (SMT) [8] constraints, and coverage-guided sampling [27].

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 [43, 23] and dynamic symbolic execution techniques [33, 66, 15, 14, 22, 48, 72, 3, 59, 2, 42, 64, 4, 62, 65, 34, 70, 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 [75, 76, 10, 38, 39, 57, 32], 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, have also been applied to hardware, where they are known as symbolic simulation [12, 13]. They can be applied both at the level of Verilog or a higher-level HDL, such as Chisel [6]. Symbolic simulation 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. 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.

Also in the context of hardware verification, a similar idea of *constrained-random verification* (CRV) [55] 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 [77, 54]. 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) [69], 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 [17]. There are approaches which can provably sample according to a desired distribution [20, 36], but are expensive to run when a large number of samples is required. Other approaches use heuristics for faster sampling [73], 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) [49] 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 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 [26] 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., ≥ 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 [61] and testing of configurable systems [60], 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 [58].

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 SMTSampler 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 [26]. 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,

[1]Available at `https://github.com/RafaelTupynamba/quicksampler/`.

arrays and uninterpreted functions. When solutions are sparse and non-linearly distributed, 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 $x > 5 \vee \phi$, where x is a 32-bit integer and ϕ 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 in 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 $x > 5 \vee \phi$, where x is a 32-bit integer and ϕ 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 ϕ 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 [31, 17, 29], or internal coverage of an SMT formula [28], 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 [29] and SMTSampler [28] algorithms we developed for efficient sampling of SAT and SMT constraints. Then, Chapter 4 describes our *coverage-guided sampling* algorithm GuidedSampler [27]. 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 $\mathit{Vars}[\phi]$ the set of variables in the formula ϕ. SAT formulas are commonly expressed using the logical operators $\wedge, \vee, \neg$. 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

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

where each of the C_i is a disjunction of literals

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

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

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \tag{2.1}$$

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 σ 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 σ. For example, one solution to the example formula (2.1) would be the assignment σ to the variables in $\mathit{Vars}[\phi] = \{x, y, z\}$ that satisfies $\sigma[\![x]\!] = \mathit{True}$, $\sigma[\![y]\!] = \mathit{False}$ and $\sigma[\![z]\!] = \mathit{False}$. We denote by $\phi[\sigma]$ the Boolean value resulting from evaluating ϕ under the assignment σ. The solutions to ϕ are the assignments σ such that $\phi[\sigma]$ is *True*. We denote by $\mathit{Sols}[\phi]$ the set of solutions to the formula. This formula ϕ has a total of $|\mathit{Sols}[\phi]| = 4$ 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 ϕ is a subset of the variables $S \subseteq \mathit{Vars}[\phi]$ 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

$$\phi = x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y) = (x \vee y) \wedge (\neg x \vee \neg y)$$

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 [40].

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 [40]. 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 ϕ as $\mathit{Vars}[\phi] = \mathit{Bool} \cup BV \cup \mathit{Array} \cup \mathit{UF}$, where Bool, BV, Array, and 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 $\wedge, \vee, \neg$. 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

$$\mathit{select}(a, 011) + v > 0110$$

contains an array variable a and a bit-vector variable v, two bit-vector constants 011 and 0110, an array function *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 $BV[n]$ 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, UF should be a set of sets, each one for each uninterpreted function type. But in this work we will just use UF to collect all the variables which are not of the previous types Bool, BV, Array, which will be collectively labeled uninterpreted functions.

Table 2.1: Types of SMT variables.

Type	Example Value
$b \in \mathit{Bool}$	$\sigma[\![b]\!] = \mathit{False}$
$v \in \mathit{BV}$	$\sigma[\![v]\!] = 01100111$
$a \in \mathit{Array}$	$\sigma[\![a]\!][x] = \begin{cases} 0110, & \text{if } x = 001 \\ 1001, & \text{if } x = 011 \\ 0101, & \text{if } x = 101 \\ 0010, & \text{otherwise} \end{cases}$
$f \in \mathit{UF}$	$\sigma[\![f]\!](x,y) = \begin{cases} 10, & \text{if } x = 0 \wedge y = 10 \\ 01, & \text{if } x = 1 \wedge y = 00 \\ 11, & \text{otherwise} \end{cases}$

Another theory common in SMT formulas is the theory of arrays, which consists of two functions *select* and *store* that satisfy the usual axiom

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

Here, $x, x' \in BV[s_x]$ are bit-vectors of a certain size s_x and $y \in BV[s_y]$ is a bit-vector with a possibly different size s_y. Here, a is an array of domain $BV[s_x]$ and range $BV[s_y]$. The function *select* returns the value at a given index from the array, while *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 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 σ. Again, we denote by $\sigma[\![v]\!]$ the concrete assignment to v under σ.

Variables in BV are fixed-size bit-vectors, such as the variable $v \in BV[8]$. Arrays must have bit-vector domains and ranges, such as the array a, with domain $BV[3]$ and range $BV[4]$. Uninterpreted functions can have any arity. The example shows the function $f : BV[1] \times BV[2] \to BV[2]$, 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 [67] and lazy [63] 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 [37].

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 [25] 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 [63]. 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 [56] 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 $\{\phi_1, \phi_2, \dots, \phi_m\}$, known as *hard constraints*, and a set of SAT formulas $\{\psi_1, \psi_2, \dots, \psi_n\}$, known as *soft constraints*, all over the same variables $V = \mathit{Vars}[\phi_i] = \mathit{Vars}[\psi_j]$, the MAX-SAT problem

$$\text{MAX-SAT}(\{\phi_1, \phi_2, \dots, \phi_m\}, \{\psi_1, \psi_2, \dots, \psi_n\})$$

is the problem of finding an assignment σ to the variables in V which satisfies all the hard constraints ϕ_i and the maximum possible number of soft constraints ψ_j.

The MAX-SMT problem has the same definition, except that the hard constraints ϕ_i and soft constraints ψ_j are allowed to be arbitrary SMT formulas including the supported theories. Whenever the conjunction of hard constraints $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_m$ 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 [52].

Our techniques do not require any particular MAX-SAT solving algorithm. The solver we used, Z3 [26], 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 $W : \mathit{Sols}[\phi] \to [0, 1]$ such that each solution $\sigma \in \mathit{Sols}[\phi]$ 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* [36], 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 $x = \mathit{True}$ are 3 times more favorable than solutions where $x = \mathit{False}$. 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 $\mathit{Sols}[\phi]$ 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 ψ_1 evaluates to *True* and predicate ψ_2 evaluates to *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 ϕ. 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 [29] and SMTSampler [28] for efficient sampling of solutions to SAT and SMT constraints.

3.1 QuickSampler Technique

Given a Boolean formula ϕ, the goal of QuickSampler is to generate unique solutions of ϕ efficiently. Another goal of QuickSampler is to make sure that solutions of ϕ 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 ϕ. The core algorithm behind QuickSampler works as follows. QuickSampler assumes that we are given an initial random solution σ (i.e., a satisfying assignment to ϕ). In our algorithm, this base solution σ is computed as the closest solution to a random assignment

$$\begin{array}{rl}
\sigma : & 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1 \\
\delta_a : & 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0 \\
\sigma_a = \sigma \oplus \delta_a : & 1\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1 \\
\delta_b : & 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 0 \\
\sigma_b = \sigma \oplus \delta_b : & 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \\
(\delta_a \vee \delta_b) : & 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0 \\
\tilde{\sigma} = \sigma \oplus (\delta_a \vee \delta_b) : & 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1
\end{array}$$

Figure 3.1: Combining two mutations over Boolean variables.

σ', as will be described later. Solution σ can be represented as a vector of 1s and 0s, where each location corresponds to a Boolean variable in ϕ and the value at that location in the vector denotes the value assigned to this variable in the solution σ. Let *Bool* be the set of all Boolean variables in ϕ. For example, in Figure 3.1 we show a possible vector σ, in a case where the number of variables is $|Bool| = 12$.

For each variable $v \in Bool$, QuickSampler finds a solution σ_v such that σ_v and σ are minimally different and $\sigma_v[\![v]\!] \neq \sigma[\![v]\!]$, where $\sigma[\![v]\!]$ is the value of the variable v in the solution σ. Note that such a solution may not exist for all variables in *Bool*. The diff between σ and σ_v, which we will denote using δ_v and which is the XOR of σ and σ_v, is called *an atomic mutation* of σ. 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 σ_a which has the first bit flipped (corresponding to variable a) and additionally other two bits flipped. The corresponding atomic mutation δ_a is also shown in Figure 3.1. Similarly, if the second variable of the formula is b, we might find a new solution σ_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 δ_b is again shown in Figure 3.1.

By definition, the atomic mutation δ_v always ensures that at least $\delta_v[\![v]\!]$ is one, i.e., σ and σ_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 σ_v given ϕ, σ, and v. Given σ, QuickSampler first computes the set of all atomic mutations by going over all the variables $v \in Bool$. Let us denote the set of all such atomic mutations by Δ^1_σ. Note that given σ and δ_v, we can compute σ_v as $\delta_v \oplus \sigma$.

After computing Δ^1_σ, QuickSampler computes sets of composite mutations Δ^k_σ for $k > 1$, where Δ^k_σ contains the bit-wise OR of all sets of k distinct mutations in Δ^1_σ. For example, if δ_a and δ_b are two mutations in Δ^1_σ such that $a \neq b$, then $\delta_a \vee \delta_b$ is a mutation present in Δ^2_σ. (Since each of δ_a and δ_b are bit-vectors, $\delta_a \vee \delta_b$ 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 $\delta_a \vee \delta_b$ is added to Δ^2_σ. If we apply the combined mutation to σ, by computing $\sigma \oplus (\delta_a \vee \delta_b)$ we obtain a new assignment $\tilde{\sigma}$, as in Figure 3.1. Note that $\tilde{\sigma}$ differs from σ on all the bits set in either of the two atomic mutations δ_a and δ_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 δ_a and δ_b consist of a minimal set of bits which can be flipped while still preserving the satisfiability of the formula. So the bits in δ_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 $\sigma \oplus (\delta_a \vee \delta_b)$, where we flip all the bits from δ_a in addition to the bits from δ_b.

In general, each mutation δ present in Δ^k_σ 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 σ to get an assignment $\tilde{\sigma}$ to the variables in ϕ. Such an assignment may or may not be a solution of ϕ. 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 ϕ. Let us denote the assignments obtained by applying all the mutations present in Δ_σ^k to σ by Σ_σ^k, i.e.,

$$\Sigma_\sigma^k = \left\{\delta \oplus \sigma \mid \delta \in \Delta_\sigma^k\right\}$$

We let $\Sigma_\sigma = \cup_{1 \leq k \leq 6} \Sigma_\sigma^k$. We found experimentally that over all benchmarks, 75% of the assignments in Σ_σ are solutions of ϕ.

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

Given a random solution σ, we described how QuickSampler generates lots of solutions that are small mutations of σ. We next describe how we generate a random solution σ. QuickSampler first chooses a random assignment σ' by picking the values of variables in *Bool* uniformly at random. Then it uses a MAX-SAT query to find a closest solution σ to the random assignment σ'. 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 σ using the method described above. Then it computes Σ_σ and outputs the elements of Σ_σ. 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 σ and also to obtain the solutions σ_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 σ, we just need to specify one hard constraint that the formula ϕ must be satisfied and $|Bool|$ soft constraints indicating that the values of each variable v should preferably be equal to their respective values in the random assignment σ', i.e., $\forall u \in Bool : \sigma[\![u]\!] = \sigma'[\![u]\!]$. In order to compute each solution σ_v, we specify two hard constraints and $|Bool| - 1$ soft constraints. The hard constraints are that the formula ϕ must be satisfied and that the value of variable v must be flipped, i.e., $\sigma_v[\![v]\!] \neq \sigma[\![v]\!]$. The soft constraints are that the values of other variables should preferably remain the same, or $\forall u \in Bool \backslash \{v\} : \ \sigma_v[\![u]\!] = \sigma[\![u]\!]$.

QuickSampler Algorithm

Algorithm 1 shows the complete QuickSampler algorithm in pseudocode. In the main QuickSampler function, for each epoch, we generate the random assignment σ' and find the closest solution σ. Then, function sample is responsible for obtaining new samples from the base solution σ. It first outputs solution σ. 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 σ_b, QuickSampler outputs it and tries to combine its corresponding atomic mutation δ_b with the previously seen atomic mutations δ_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 σ_a and σ_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 in epochs, with each epoch being associated with a base solution σ. The main sampling function QuickSampler(ϕ, S) works by repeatedly calling sample$(\sigma, \phi, S, \mathit{unsatVars})$ to perform one epoch of sampling. For each epoch, we choose a base solution σ as follows. First, we choose a random assignment σ' of variables in S, covering the whole space uniformly. Then, function getConditions(ϕ, σ', 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 σ'. Finally, we use a MAX-SAT query to find the closest solution σ 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$(\sigma, \phi, S, \mathit{unsatVars})$ implements one epoch of the sampling algorithm as follows. First, we output the base solution σ. We again use getConditions(ϕ, σ, S) to collect the set $C_\sigma = \{v = \sigma[\![v]\!] \mid v \in S\}$ of conditions which are true for assignment σ. Then, for each variable v in the independent support S, we use a MAX-SAT query to find a new solution σ_b which is as close as possible to σ, 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 σ_b is found, we output it and create a map *newMutations* of new samples which were discovered using σ_b. We add σ_b to *newMutations* as a new sample of level 1 and, for every sample σ_a of level $l < 6$ in *mutations*, we combine σ_a with σ_b, generating a new sample $\tilde{\sigma}$. If $\tilde{\sigma}$ is a previously unknown sample (not present

Algorithm 1 QuickSampler Algorithm.

```
function QuickSampler($\phi, S$)
    $unsatVars \leftarrow \{\}$
    while not done do
        $\sigma' \leftarrow$ generateRandomAssignment($\phi, S$)
        $C_{\sigma'} \leftarrow$ getConditions($\phi, \sigma', S$)
        $\sigma \leftarrow$ MAX-SAT($\{\phi\}, C_{\sigma'}$)
        if not $\sigma$ then break
        sample($\sigma, \phi, S, unsatVars$)

function sample($\sigma, \phi, S, unsatVars$)
    output($\{\sigma\}$)
    $C_\sigma \leftarrow$ getConditions($\phi, \sigma, S$)
    $neighbors \leftarrow \{\}$
    $mutations \leftarrow$ new $map(assignment \rightarrow int)$
    for $v$ in $S$ where $v \notin unsatVars$ do
        $\sigma_b \leftarrow$ MAX-SAT($\{\phi,\ v \neq \sigma[\![v]\!]\}, C_\sigma \backslash \{v = \sigma[\![v]\!]\}$)
        if $\sigma_b$ then
            if $\sigma_b \notin neighbors$ then
                $neighbors \leftarrow neighbors \cup \{\sigma_b\}$
                $newMutations \leftarrow$ new $map(assignment \rightarrow int)$
                $newMutations[\sigma_b] = 1$
                output($\{\sigma_b\}$)
                for $\sigma_a$ in $mutations$ where $mutations[\sigma_a] < 6$ do
                    $\tilde{\sigma} \leftarrow$ combine($\sigma, \sigma_a, \sigma_b, S$)
                    if $\tilde{\sigma} \notin mutations.keys() \cup newMutations.keys()$ then
                        $newMutations[\tilde{\sigma}] = mutations[\sigma_a] + 1$
                        output($\{\tilde{\sigma}\}$)
                for $\tilde{\sigma}$ in $newMutations.keys()$ do
                    $mutations[\tilde{\sigma}] \leftarrow newMutations[\tilde{\sigma}]$
        else
            $unsatVars \leftarrow unsatVars \cup \{v\}$

function combine($\sigma, \sigma_a, \sigma_b, S$)
    $\tilde{\sigma} \leftarrow$ new $assignment()$
    for $v$ in $S$ do
        $\delta_a \leftarrow \sigma[\![v]\!] \oplus \sigma_a[\![v]\!]$
        $\delta_b \leftarrow \sigma[\![v]\!] \oplus \sigma_b[\![v]\!]$
        $\tilde{\sigma}[\![v]\!] \leftarrow \sigma[\![v]\!] \oplus (\delta_a \vee \delta_b)$
    return $\tilde{\sigma}$
```

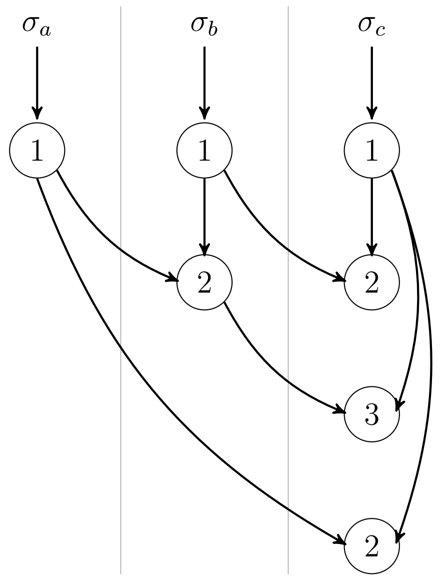


Figure 3.2: Eager combination of mutations.

in *mutations* or *newMutations*), we output it and add it to *newMutations* as a new sample of level $l + 1$. Finally, the samples from *newMutations* are merged into *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.

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 *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 σ_a is returned by the solver, we learn one atomic mutation δ_a, represented by the first circle in the figure. Then, as soon as solution σ_b becomes available, we learn the atomic mutation δ_b and also combine it with δ_a to generate a mutation in Δ^2_σ. Then, as soon as solution σ_c

becomes available, we learn the mutation δ_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 σ_b is found, it is immediately combined with the previously known samples σ_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 σ_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 [17], 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 *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 COMBINE, we only need to assign values to the variables in S.

Unsatisfiable Variables

If one MAX-SAT query 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 *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 *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($\phi$)
2:     while not done do
3:         $\sigma' \leftarrow$ generateRandomAssignment($\phi$)
4:         $C_{\sigma'} \leftarrow$ getConditions($\phi, \sigma'$)
5:         $\sigma \leftarrow$ MAX-SMT($\{\phi\}, C_{\sigma'}$)
6:         output($\{\sigma\}$)
7:         $\Sigma^1_\sigma \leftarrow$ computeNeighboringSolutions($\phi, \sigma$)
8:         output($\Sigma^1_\sigma$)
9:         $\alpha \leftarrow 1, k \leftarrow 1, \Sigma_\sigma \leftarrow \Sigma^1_\sigma$
10:        while $\alpha \geq \alpha_{min} \wedge k < 6$ do
11:            $(\Sigma^{k+1}_\sigma, \alpha, \Sigma_\sigma) \leftarrow$ combineMutations($\Sigma^k_\sigma, \Sigma^1_\sigma, \Sigma_\sigma, \phi$)
12:            output($\Sigma^{k+1}_\sigma$)
13:            $k \leftarrow k + 1$
14:
15: function computeNeighboringSolutions($\phi, \sigma$)
16:     $C_\sigma \leftarrow$ getConditions($\phi, \sigma$)
17:     $\Sigma^1_\sigma \leftarrow \{\}$
18:     for $c$ in $C_\sigma$ do
19:         $\tilde{\sigma} \leftarrow$ MAX-SMT($\{\phi, \neg c\}, C_\sigma \backslash \{c\}$)
20:         if $\tilde{\sigma}$ then
21:             $\Sigma^1_\sigma \leftarrow \Sigma^1_\sigma \cup \{\tilde{\sigma}\}$
22:     return $\Sigma^1_\sigma$
23:
24: function combineMutations($\Sigma^k_\sigma, \Sigma^1_\sigma, \Sigma_\sigma, \phi$)
25:     $valid \leftarrow 0, checks \leftarrow 0$
26:     for $(\sigma_a, \sigma_b)$ in $\Sigma^k_\sigma \times \Sigma^1_\sigma$ do
27:         $\tilde{\sigma} \leftarrow \Psi_\sigma(\sigma_a, \sigma_b)$
28:         if $\tilde{\sigma} \notin \Sigma_\sigma$ then
29:             $\Sigma_\sigma \leftarrow \Sigma_\sigma \cup \{\tilde{\sigma}\}$
30:             $checks \leftarrow checks + 1$
31:             if $\phi[\tilde{\sigma}]$ then
32:                 $\Sigma^{k+1}_\sigma \leftarrow \Sigma^{k+1}_\sigma \cup \{\tilde{\sigma}\}$
33:                 $valid \leftarrow valid + 1$
34:     return $(\Sigma^{k+1}_\sigma, valid/checks, \Sigma_\sigma)$
```

Algorithm 2 presents the main SMTSampler procedure, which takes as input an SMT formula ϕ. Just like QuickSampler, the SMTSampler algorithm works over several epochs. In each epoch, we first sample one initial base solution σ. This is done by generating a random assignment σ' to the variables of the formula in line 3 and then using MAX-SMT to obtain the solution σ which is closest to σ' in line 5. Then, in line 7, we use the function computeNeighboringSolutions to compute a set Σ_σ^1 of *neighboring solutions* for σ. 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* $\Psi : X \times X \times X \to X$, where X is the space of all possible assignments to the variables $Vars[\phi]$ in the formula. We denote by $\Psi_\sigma(\sigma_a, \sigma_b)$ the application of the combination function to the base solution σ and two other solutions σ_a and σ_b. Intuitively, the combination function Ψ computes the mutations which can be applied to σ to generate σ_a and σ_b, then merges those two mutations together to produce a new assignment. In QuickSampler, the combination function would be defined as

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

For SMTSampler, we provide a natural extension of this definition that can handle bit-vectors, arrays and uninterpreted functions. The assignment returned by Ψ 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 Ψ to combine those changes in an additive way. The full definition of Ψ will be presented later.

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

Now assume that we already constructed a set Σ_σ^k. We can inductively build the set Σ_σ^{k+1} as follows. For each pair of samples $\sigma_a \in \Sigma_\sigma^k$ and $\sigma_b \in \Sigma_\sigma^1$, we apply the combination function Ψ to generate a new sample $\tilde{\sigma} = \Psi_\sigma(\sigma_a, \sigma_b)$. If $\tilde{\sigma}$ is an element of Σ_σ, it has already been checked and is discarded. Otherwise, we add it to Σ_σ 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 Σ_σ^{k+1}.

Note that, unlike QuickSampler, in SMTSampler we do not use a strategy of eager generation of samples. Instead, we compute the sets $\Sigma_\sigma^1, \Sigma_\sigma^2, \ldots$ in order. We chose this approach because it allows an adaptive generation of samples based on accuracy. During the construction of Σ_σ^{k+1} from Σ_σ^k, we keep statistics on which fraction α of the checked samples were valid. If this fraction is below a certain threshold α_{min}, such as 0.1, we do not generate

Σ_σ^{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 $\cup_{0 \leq k \leq 6} \Sigma_\sigma^k$, 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 σ for each epoch is computed similarly to QuickSampler. We first generate a random assignment σ' 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 σ', because we do not know initially which indices will be relevant for those variables. After generating σ', we choose σ as a solution which is as close as possible to σ'. This is done to explore as much of the solution space as possible, generating base solutions σ from different parts of the space.

The problem of finding a solution σ which is as close as possible to σ' 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 ϕ 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 σ'. Analogously, we add one soft constraint $b = \sigma'[\![b]\!]$ for each Boolean variable b.

Computing Atomic Mutations

After generating a base solution σ, we first compute a set of neighboring solutions of the base solution σ, 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_σ which are true for σ. 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_σ, 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.

Type	Example Condition
$b \in Bool$	$b = False$
$v \in BV$	$extract(v, 5) = 1$
$a \in Array$	$extract(a[011], 3) = 1$
$f \in UF$	$extract(f(0, 10), 1) = 0$

Constructing C_σ. Function GETCONDITIONS produces C_σ 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 $extract(v, i) = extract(\sigma[\![v]\!], i)$, 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 $extract\ (a[x], i) = extract\ (\sigma[\![a]\!][x], i)$. 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 Σ^1_σ. 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$(\{\phi, \neg c\}, C_\sigma \backslash \{c\})$ 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 $extract(v, 0) = 0$ and $extract(v, 1) = 0$. 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.

$$\begin{aligned}
v :&\quad 1\,1\,0\,0\,0\,1\,0\,1\\
\delta_a = v \oplus v_a :&\quad 1\,0\,0\,0\,0\,1\,1\,0\\
v_a :&\quad 0\,1\,0\,0\,0\,0\,1\,1\\
\delta_b = v \oplus v_b :&\quad 0\,0\,0\,1\,0\,1\,0\,0\\
v_b :&\quad 1\,1\,0\,1\,0\,0\,0\,1\\
(\delta_a \vee \delta_b) :&\quad 1\,0\,0\,1\,0\,1\,1\,0\\
\psi_v(v_a, v_b) = v \oplus (\delta_a \vee \delta_b) :&\quad 0\,1\,0\,1\,0\,0\,1\,1
\end{aligned}$$

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

Combining Mutations

Now we define the combination function Ψ which is used to generate new samples. Assume that we already know the base solution σ and two additional solutions to the formula σ_a and σ_b, which are close to σ. Those additional solutions can be obtained by calling computeNeighboringSolutions or they could be already generated by an application of the Ψ function.

The combination function Ψ, which combines entire solutions, is constructed by defining a method ψ to combine the values of each of the variables in the formula. We define

$$\Psi_\sigma(\sigma_a, \sigma_b)[\![v]\!] = \psi_{\sigma[\![v]\!]}\left(\sigma_a[\![v]\!], \sigma_b[\![v]\!]\right)$$

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

Next, we define how the combination method ψ 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 δ_a and δ_b indicate exactly which bits differ between the base value and each of the additional values. One can think of those differences as mutations that 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 δ_a to v. For bit-vectors, we define a combined mutation through the OR operator, producing $(\delta_a \vee \delta_b)$. This resulting mutation can be applied to the base value v, producing a new value $v \oplus (\delta_a \vee \delta_b)$. Thus, for bit-vectors, ψ is defined as

$$\psi_v(v_a, v_b) = v \oplus ((v \oplus v_a) \vee (v \oplus v_b))$$

For Boolean values, we use the same technique: $\psi_b(b_a, b_b) = b \oplus ((b \oplus b_a) \vee (b \oplus b_b))$. This way, Boolean values behave the same as bit-vectors of size 1.

Now we define how to apply the combination method ψ 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 α has explicitly defined values for indices in the set $I(\alpha)$, and a default value $d(\alpha)$ for all other indices. Arrays α_a and α_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

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

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 ψ and, consequently, Ψ.

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 σ_a from σ and the mutation that generates σ_b from σ and then combines those two mutations in an additive way. If σ_a and σ_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 $\Psi_\sigma\left(\Psi_\sigma(\sigma_a, \sigma_b), \sigma_c\right) = \Psi_\sigma\left(\sigma_a, \Psi_\sigma(\sigma_b, \sigma_c)\right)$. In SMTSampler, the samples generated in Σ_σ^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 [27], 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 ϕ be a satisfiable SMT formula (i.e., $\mathit{Sols}[\phi] \neq \varnothing$) over variables $V = \mathit{Vars}[\phi]$ and let $\psi_1, \psi_2, \ldots, \psi_n$ be n SMT formulas which only include variables from V (i.e., $\mathit{Vars}[\psi_i] \subseteq V$). The formulas ψ_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 ϕ over the coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$, as follows:

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

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

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

For our example formula $\phi = b \vee (x + 2 > y)$, if we have coverage predicates $\psi_1 = b$ and $\psi_2 = x + 2 > y$, then $G_{(\mathit{False}, \mathit{False})} = \varnothing$ 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 ϕ with coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$ is a random process that returns each possible solution $\sigma \in \mathit{Sols}[\phi]$ 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 ϕ, 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 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 ϕ 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 GuidedSampler($\phi, \underline{\{\psi_1, \psi_2, \dots, \psi_n\}}$).

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

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 σ (lines 3-7), instead of generating σ based only on the random assignment σ'.

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 σ 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 σ' to the variables of the formula in lines 3-6 and using a MAX-SMT [56] solver call to obtain a solution σ which is closest to G_b and to σ' 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 Σ^1_σ of *neighboring solutions* to σ that belong to different coverage classes than σ. This function will also be described later.

For combining mutations, we leverage the same combination function Ψ constructed in Section 3.2 for SMTSAMPLER. Again, the *combination function* is defined as $\Psi : X \times X \times X \to X$, where X is the space of all possible assignments to the variables in $V = \mathit{Vars}[\phi]$. We denote by $\Psi_\sigma(\sigma_a, \sigma_b)$ the application of the combination function to the base solution σ and two other solutions σ_a and σ_b. Intuitively, the combination function Ψ computes the mutations which can be applied to σ to generate σ_a and σ_b, then merges those two mutations together to produce a new assignment. The assignment returned by Ψ 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 Ψ to combine those changes in an additive way.

We next describe how function COMBINEMUTATIONS called by GUIDEDSAMPLER in line 13 uses Ψ to generate new samples. We denote by Σ^1_σ the set of neighboring solutions to σ obtained from COMPUTENEIGHBORINGSOLUTIONS. Starting from Σ^1_σ, our goal is to compute sets Σ^k_σ which will contain solutions generated by combining k atomic mutations, for $1 \leq k \leq 6$. Throughout the current epoch, we maintain a set Σ_σ of samples which were computed so far, both valid and invalid. Initially, $\Sigma_\sigma = \Sigma^1_\sigma$. We also maintain a set Γ_σ of all the coverage classes for which we have generated a solution in the current epoch.

Now assume that we already constructed a set Σ^k_σ. We can inductively build the set Σ^{k+1}_σ 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 Ψ to generate a new sample $\tilde{\sigma} = \Psi_\sigma(\sigma_a, \sigma_b)$. If $\tilde{\sigma}$ is an element of Σ_σ, it has already been checked and is discarded. Otherwise, we add it to Σ_σ 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 Σ_{σ}^{k+1}.

Just like SMTSampler, during the construction of Σ_{σ}^{k+1} from Σ_{σ}^{k}, we also keep statistics on which fraction α 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 $\cup_{0 \leq k \leq 6} \Sigma_{\sigma}^{k}$, 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.

Computing the Base Solution

Now, we describe how the initial base solution σ 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 σ' by choosing values to the Boolean and bit-vector variables in the formula uniformly at random. After choosing G_b and σ', we want to find a solution σ which is as close as possible to the coverage class G_b and to the random assignment σ'. This is done to explore as much of the coverage classes and the solution space as possible.

The problem of finding a solution σ which is as close as possible to G_b and σ' can be encoded as a MAX-SMT [56] optimization problem to be solved by the constraint solver. We add one hard constraint stating that the formula ϕ 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 ψ_i should be equal to the random Boolean b_i. For each bit-vector or Boolean variable $v \in \mathit{Vars}[\phi]$, we add one soft constraint to $C_{\sigma'}$ stating that variable v should have the same value that it has in σ'.

Computing Atomic Mutations

After generating a base solution σ, we compute neighboring solutions of the base solution σ, 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 σ. We want our neighboring solutions to flip the value of some coverage predicates. The first step is collecting a set of conditions S_b which are true about the coverage predicates for the base solution σ. 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 ϕ, 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 σ. 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_σ. Function getConditions produces C_σ by collecting conditions for each variable in the formula. The conditions can be, for example, $\mathit{extract}(v, 5) = 1$, 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 Σ^1_σ. After collecting the conditions in S_b and C_σ, we want to compute neighboring solutions by picking one condition $c \in S_b$ and using the constraint solver to find a solution to $\phi \wedge \neg c$. However, the neighboring solution should be as similar as possible to σ. We express such constraint by requiring that the new solution should satisfy the maximum possible number of the remaining conditions in $S_b \backslash \{c\}$ and C_σ. 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 ϕ that does not satisfy c. And we also specify as soft constraints the conditions in $S_b \backslash \{c\}$ and in C_σ.

Combining Mutations

	Bits	*Predicates*
v :	1 1 0 0 0 1 0 1	0 1 0
v_a :	0 1 0 0 0 0 1 1	1 1 0
v_b :	1 1 0 1 0 0 0 1	0 0 1
$\delta_a = v \oplus v_a$:	1 0 0 0 0 1 1 0	1 0 0
$\delta_b = v \oplus v_b$:	0 0 0 1 0 1 0 0	0 1 1
$(\delta_a \vee \delta_b)$:	1 0 0 1 0 1 1 0	1 1 1
$\psi_v(v_a, v_b) = v \oplus (\delta_a \vee \delta_b)$:	0 1 0 1 0 0 1 1	1 0 1

Figure 4.1: Combining two mutations over bit-vectors of size 8. We also list values for 3 coverage predicates ψ_1, ψ_2, ψ_3.

The combination function Ψ 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 ψ_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.

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 [26] as the underlying solver. QuickSampler uses the Z3 optimization subsystem νZ [9] in order to solve the MAX-SAT queries. We also use the *push*() and *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 `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 [17] and SearchTreeSampler [31]. 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 [17] available online[2]. From the benchmarks listed in [17], 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[3], UniGen2 reported an error because the specified independent sup-

[2]Benchmarks and source code for UniGen2 were obtained from `https://bitbucket.org/kuldeepmeel/unigen`.

[3]GuidanceService2.sk_2_27, GuidanceService.sk_4_27, IssueServiceImpl.sk_8_30, PhaseService.sk_14_27, ActivityService.sk_11_27, IterationService.sk_12_27, ActivityService2.sk_10_27, ConcreteActivityService.sk_13_28, NotificationServiceImpl2.sk_10_36, LoginService.sk_20_34.

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

Atomic mutations	Total	Valid	%
0	1	1	100%
1	32	32	100%
2	511	492	96%
3	5619	5208	93%
4	47493	42179	89%
5	346367	282860	82%
6	2143385	1571553	73%
Total	2543409	1902325	75%

port is not really an independent support for the formula. In all those benchmarks, we verified that the number of solutions computed by the exact model counter sharpSAT [71] is larger than $2^{|S|}$, which should not happen if S is a real independent support for the formula. So we decided to exclude those benchmarks from our results. The results in this section include the remaining 163 benchmarks.

On 3 benchmarks, UniGen2 could not estimate the number of solutions: on parity.sk_11_11, UniGen2 raised a floating-point exception and on isolateRightmost.sk_7_481 and listReverse.sk_11_43, the ApproxMC [19] model counter used by UniGen2 couldn't finish even in 40 hours. On 2 benchmarks, UniGen2 estimated the number of solutions but couldn't produce any samples: on doublyLinkedList.sk_8_37 it timed out and on diagStencilClean.sk_41_36 it ran out of memory.

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 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 QuickSampler, SearchTreeSampler and 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 QuickSampler. We additionally define t_q^* to be the estimated time per valid sample that 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 QuickSampler. We have included the largest benchmarks (more than 4000 variables), the benchmarks which were listed as representative benchmarks in [17] and the benchmarks used for uniformity plots, which will be presented later. This includes the benchmarks which QuickSampler or 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 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 UniGen2 failed completely to compute the number of solutions. The second group of columns shows some results for 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 QuickSampler, we repeat the information about the number of samples generated and fraction of samples which are valid, and additionally include the

Table 5.2: Benchmarks for evaluation of QuickSampler.

					QuickSampler			
Benchmark	$\|S\|$	Vars	Clauses	Solutions	Epochs	Calls	Samples	Valid
blasted_case47	28	118	328	262144	244	6616	10010929	0.564
blasted_case110	17	287	1263	16384	1387	22208	10001202	0.822
s820a_7_4	23	616	1703	591872	128	3093	10002673	0.770
s820a_15_7	23	685	1987	722944	114	2759	10014350	0.674
s1238a_3_2	32	686	1850	2466250752	9	328	10140047	0.936
s1196a_3_2	32	690	1805	1038090240	11	393	10077447	0.803
s832a_15_7	23	693	2017	3713024	83	2014	10017640	0.818
blasted_case_1_b12_2	45	827	2725	274877906944	1	89	10021799	0.739
blasted_squaring16	72	1627	5835	1865275930882	0	65	10304220	0.209
blasted_squaring7	72	1628	5837	274408144896	0	68	11344920	0.112
70.sk_3_40	40	4670	15864	8589934592	8	304	10134785	1.000
ProcessBean.sk_8_64	64	4768	14458	≈7009386627072	1	86	10011221	0.906
56.sk_6_38	38	4842	17828	3690987520	9	334	10049283	0.930
35.sk_3_52	52	4915	10547	4398046511104	2	95	10717156	1.000
80.sk_2_48	48	4969	17060	1099511627776	2	126	10252598	1.000
7.sk_4_50	50	6683	24816	2199023255552	2	124	10139607	1.000
doublyLinkedList.sk_8_37	37	6890	26918	2038431744	106	3425	10003513	0.267
19.sk_3_48	48	6993	23867	2959802892288	1	89	10198861	0.937
29.sk_3_45	45	8866	31557	347892350976	2	120	10045170	0.855
isolateRightmost.sk_7_481	481	10057	35275	-	0	59	11191269	0.878
17.sk_3_45	45	10090	27056	274877906944	3	157	10181716	1.000
81.sk_5_51	51	10775	38006	18141941858304	1	52	11099585	0.867
LoginService2.sk_23_36*	36	11511	41411	≈163840	272	6019	10001533	0.724
sort.sk_8_52	52	12125	49611	≈88046829568	2	105	10563617	0.625
parity.sk_11_11*	11	13116	47506	-	68	615	3833	0.809
77.sk_3_44	44	14535	27573	18253611008	6	249	10014904	0.966
20.sk_1_51	51	15475	60994	37108517437440	1	52	11126152	0.910
enqueueSeqSK.sk_10_42	42	16466	58515	≈3355443200	4	207	10008980	0.762
karatsuba.sk_7_41*	41	19594	82417	≈1245184	2	86	670641	0.088
diagStencilClean.sk_41_36*	36	378131	2110471	≈13	5	66	87	0.701
tutorial3.sk_4_31*	31	486193	2598178	≈49283072	6	193	2114947	0.798

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 $\log_{10}(t_s/t_q)$ 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 $t_s/t_q = 6.6\cdot10^{-5}$. 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.

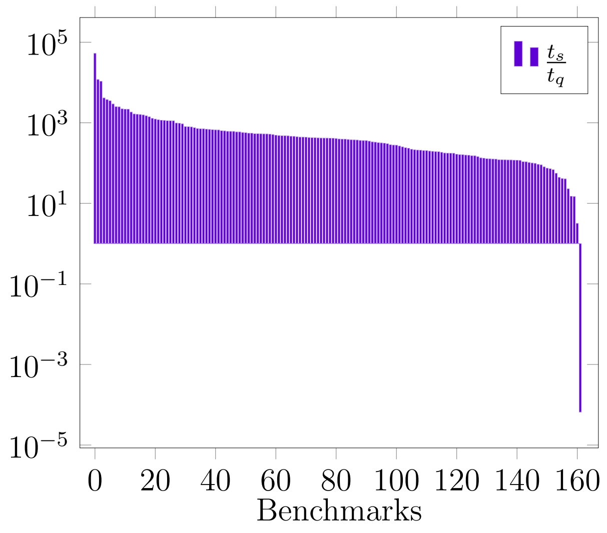
Table 5.3: Comparison of SAT sampling algorithms.

	QuickSampler				SearchTreeSampler		UniGen2	
Benchmark	Samples	Valid	t_q (μs)	t_q^* (μs)	Samples	t_s/t_q	Samples	t_u/t_q
blasted_case47	10010929	0.564	7.5	26	11694350	41.3	3932170	426
blasted_case110	10001202	0.822	28.3	29	8502350	14.9	245762	34
s820a_7_4	10002673	0.770	5.9	34	4007950	151.6	2959363	802
s820a_15_7	10014350	0.674	9.0	66	3875900	103.2	3614721	674
s1238a_3_2	10140047	0.936	2.7	211	1917850	707.2	1001	60515
s1196a_3_2	10077447	0.803	3.2	246	1848850	609.1	1001	60320
s832a_15_7	10017640	0.818	6.4	100	2742600	204.4	1001	3803
blasted_case_1_b12_2	10021799	0.739	2.9	305	1001600	1235.7	1001	71769
blasted_squaring16	10304220	0.209	15.8	1961	285450	799.7	1001	215680
blasted_squaring7	11344920	0.112	32.1	3788	255750	438.1	1001	22186
70.sk_3_40	10134785	1.000	5.8	1236	4091950	151.2	1001	109854
ProcessBean.sk_8_64	10011221	0.906	4.1	1294	297900	2932.3	1001	179418
56.sk_6_38	10049283	0.930	5.3	694	1685350	406.3	1001	71623
35.sk_3_52	10717156	1.000	2.3	229	2348300	664.6	1001	435883
80.sk_2_48	10252598	1.000	4.0	1399	2572650	350.5	1001	103909
7.sk_4_50	10139607	1.000	4.9	1778	1717250	429.5	1001	296687
doublyLinkedList.sk_8_37	10003513	0.267	678.4	6308	231850	22.9	0	-
19.sk_3_48	10198861	0.937	4.1	2010	756400	1156.1	1001	814253
29.sk_3_45	10045170	0.855	6.7	2772	215450	2483.0	1001	1995316
isolateRightmost.sk_7_481	11191269	0.878	11.3	3293	6000	52789.2	0	-
17.sk_3_45	10181716	1.000	5.7	2374	1600150	392.8	1001	3207452
81.sk_5_51	11099585	0.867	4.0	3863	75850	11859.7	1001	1035125
LoginService2.sk_23_36*	10001533	0.724	680.3	3212	1593200	14.8	778250	34
sort.sk_8_52	10563617	0.625	31.1	7354	30650	3775.2	1001	155253
parity.sk_11_11*	3833	0.809	2322699.9	3535813	462	3.2	0	-
77.sk_3_44	10014904	0.966	5.8	1580	1478300	420.4	1001	2552683
20.sk_1_51	11126152	0.910	4.0	3751	84250	10695.1	1001	2360454
enqueueSeqSK.sk_10_42	10008980	0.762	34.8	21412	29450	3512.4	1001	30830
karatsuba.sk_7_41*	670641	0.088	125504.0	203615	50	1116.2	1001	61
diagStencilClean.sk_41_36*	87	0.701	120336466	120397476	908868	0.000066	0	-
tutorial3.sk_4_31*	2114947	0.798	4281.2	362747	1200	693.2	506	18783

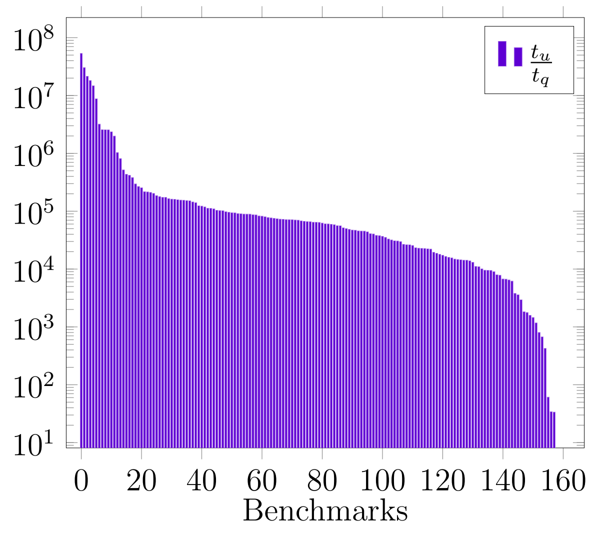
The opposite effect can be seen, for example, on parity.sk_11_11, where MiniSAT was only able to complete a small number of solver calls.

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



(a) SearchTreeSampler/QuickSampler.



(b) UniGen2/QuickSampler.

Figure 5.1: Average time per valid sample.

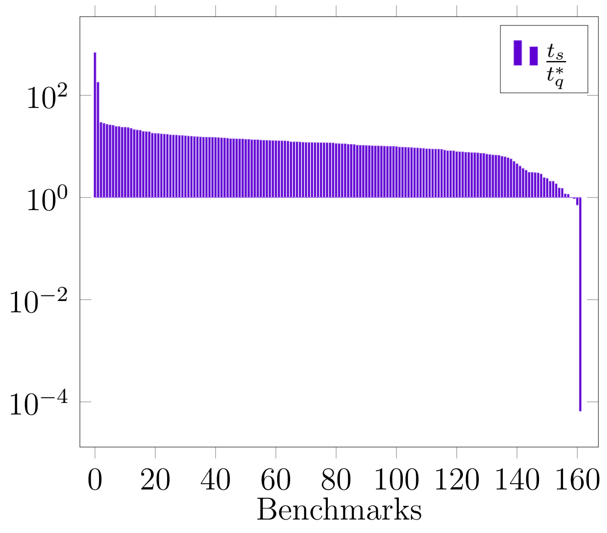
Table 5.4: Mean ratio of sampling speeds across all benchmarks.

Ratio	Mean
t_s/t_q	$10^{2.5\pm0.8}$
t_u/t_q	$10^{4.7\pm1.0}$
t_s/t_q^*	$10^{1.0\pm0.5}$
t_u/t_q^*	$10^{3.2\pm0.7}$
u_q/u_s	$10^{2.3\pm0.7}$
u_q/u_u	$10^{4.4\pm1.1}$

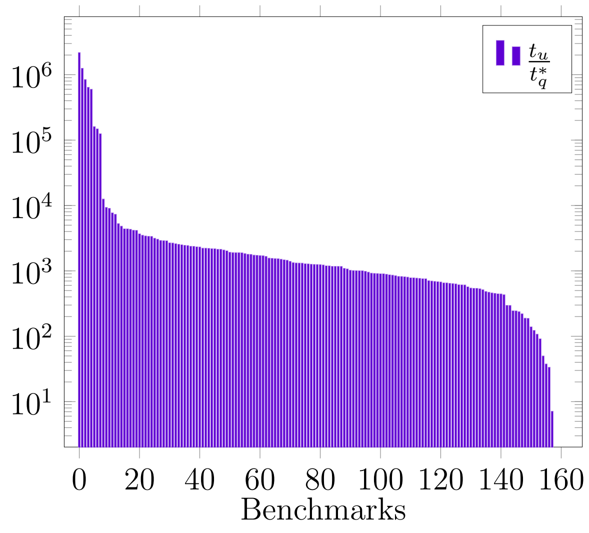
an experiment to evaluate the number of unique valid solutions generated.

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 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 QuickSampler and SearchTreeSampler on the number of

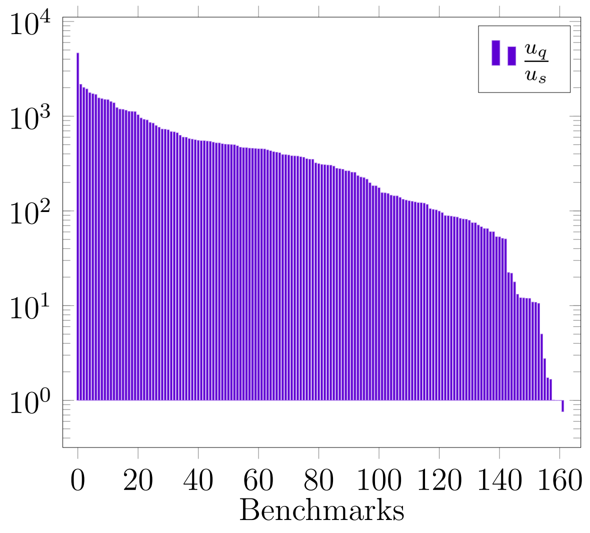


(a) SearchTreeSampler/QuickSampler.

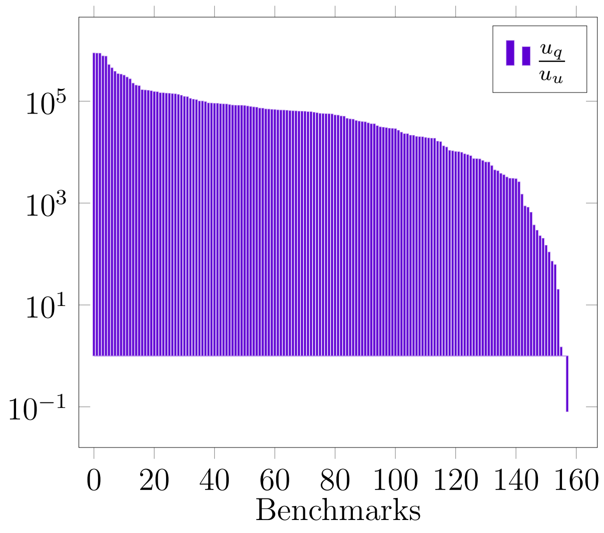


(b) UniGen2/QuickSampler.

Figure 5.2: Average time per valid sample, including time to check validity.



(a) QuickSampler/SearchTreeSampler.



(b) QuickSampler/UniGen2.

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 χ^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-

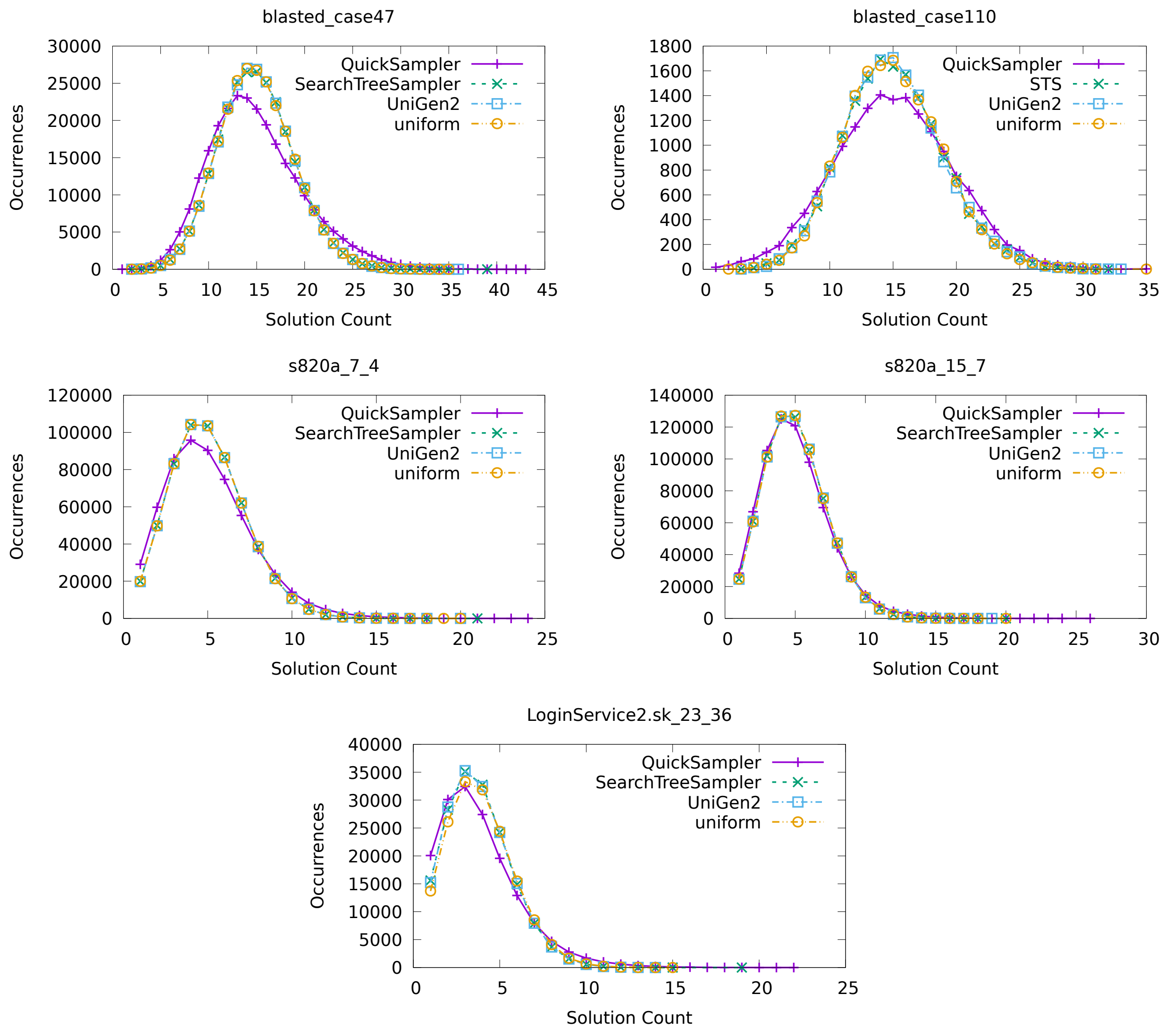


Figure 5.4: Histograms comparing the distribution of solutions.

Table 5.5: Chi-squared uniformity test.

	Not Rejected	Rejected
QuickSampler	149	11
SearchTreeSampler	153	7
UniGen2	155	5

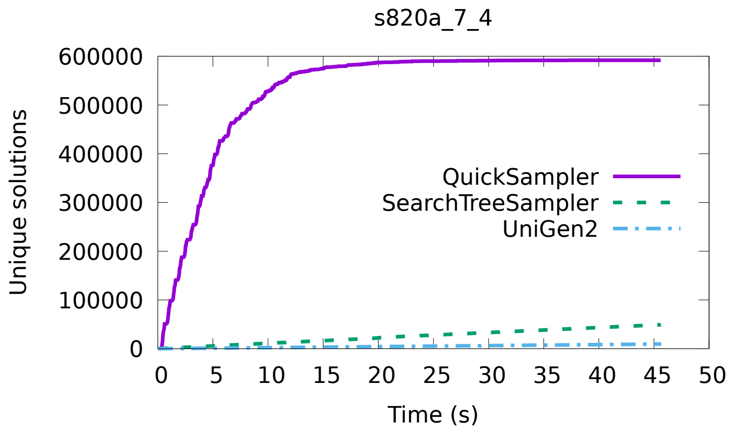


Figure 5.5: s820a_7_4 unique solutions.

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 SearchTreeSampler and UniGen2 are more uniform, but QuickSampler 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 QuickSampler 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 QuickSampler 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 SearchTreeSampler, the average was 0.996 and standard deviation 0.038. SearchTreeSampler also performed the worst on the benchmark doublyLinkedList.sk_8_37, with value 0.538, and all other benchmarks having $u/\bar{u} > 0.92$. UniGen2 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, UniGen2 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]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.

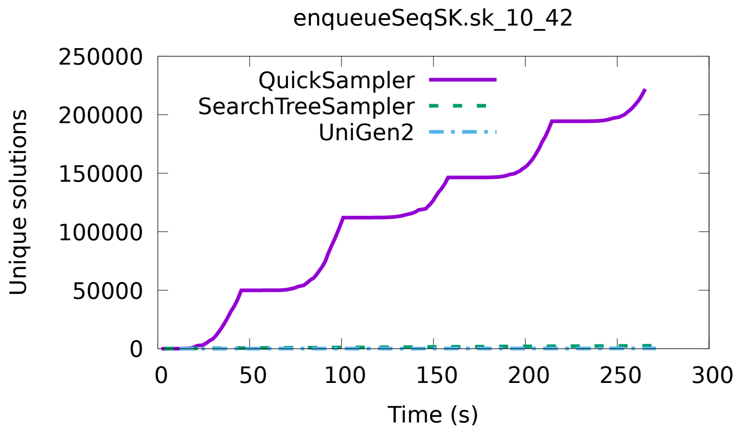


Figure 5.6: enqueueSeqSK.sk_10_42 unique solutions.

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 [26] 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`.

Table 5.6: Z3 tactics for conversion into SAT.

Tactic	Description
`simplify`	Simplify the formula
`bvarray2uf`	Encode arrays as UFs
`ackermannize_bv`	Apply Ackermann's encoding to UFs
`bit-blast`	Bit-blast the bit-vector variables

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 [45] 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× 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 $\mathit{select}(\mathit{store}(\ldots), \ldots)$ 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 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

Table 5.7: Average benchmark statistics for SMTSampler.

Benchmarks	n	nodes	$\|Array\|$	$\|BV\|$	$\|Bool\|$	bits	$\|UF\|$
QF_AUFBV/ecc	4	291	1	42	12	2785	1
QF_ABV/bmc-arrays	3	855	1	1	0	53	0
QF_ABV/stp_samples	15	1139	1	24	0	192	0
QF_ABV/dwp_formulas	5	613	3	32	0	428	0
QF_ABV/egt	15	90	1	0	0	0	0
QF_ABV/bench_ab	15	317	1	0	0	6	0
QF_ABV/platania/...member	12	4152	36	463	0	14816	0
QF_BV/bmc-bv	10	782	0	13	0	422	0
QF_BV/bmc-bv-svcomp14	8	7518	0	205	1055	7607	0
QF_BV/spear/zebra_v0.95a	9	571	0	185	0	2012	0
QF_BV/RWS	9	1086	0	21	0	3628	0
QF_BV/gulwani-pldi08	5	1146	0	130	0	950	0
QF_BV/stp_samples	14	793	0	22	0	200	0
QF_BV/brummayerbiere2	3	632	0	2	0	149	0
QF_BV/tacas07	3	8812	0	345	588	16620	0
QF_BV/bench_ab	13	23	0	2	0	41	0
QF_BV/sage/app2	13	240	0	25	0	211	0
QF_BV/sage/app9	8	271	0	35	0	391	0
QF_BV/sage/app8	15	978	0	93	0	1047	0
QF_BV/sage/app5	12	269	0	29	0	355	0
QF_BV/sage/app1	10	117	0	21	0	271	0
QF_BV/sage/app12	12	247	0	31	0	358	0

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.

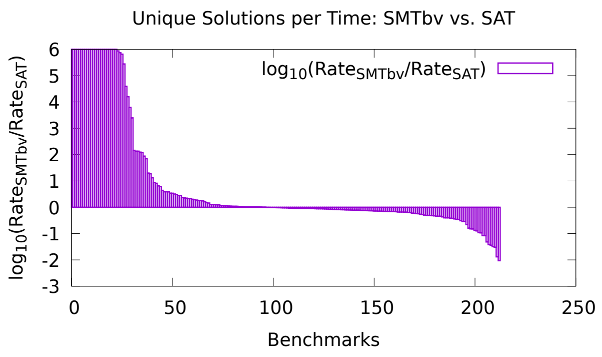
Then Table 5.8 presents average results from the experiments with the three techniques. First, we list the number of unique solutions produced, then the ratio of unique solutions over time and, finally, the total coverage obtained. Those values are also averages over the n benchmarks in each directory. 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.

Benchmarks	Unique solutions SMTbv	Unique solutions SMTbit	Unique solutions SAT	Unique solutions / second SMTbv	Unique solutions / second SMTbit	Unique solutions / second SAT	Coverage SMTbv	Coverage SMTbit	Coverage SAT
QF_AUFBV/ecc	209535	26860	49087	579	748	680	407	856	596
QF_ABV/bmc-arrays	807570	1229174	1096836	1347	1757	1085	3090	3480	3134
QF_ABV/stp_samples	258440	437702	287537	179	309	220	4484	4768	4035
QF_ABV/dwp_formulas	898530	1300388	319558	746	989	259	1276	1016	377
QF_ABV/egt	769975	916283	1333783	879	1093	2114	138	136	140
QF_ABV/bench_ab	181716	341169	689368	761	1295	2621	129	129	114
QF_ABV/platania/..member	2085	113046	0	558	515	0	83	44	0
QF_BV/bmc-bv	439867	1250125	297913	6161	7011	5604	98	94	95
QF_BV/bmc-bv-svcomp14	40809	5915	131089	194	212	153	3081	3108	3655
QF_BV/spear/zebra_v0.95a	196822	18633	35	858	1319	216	530	534	28
QF_BV/RWS	4776	476	201	31	71	23	4766	4766	2517
QF_BV/gulwani-pldi08	167745	153184	127868	268	305	279	2113	2150	2107
QF_BV/stp_samples	505214	501507	438752	375	373	399	1426	1331	969
QF_BV/brummayerbiere2	263405	531815	712535	362	625	857	224	187	224
QF_BV/tacas07	33928	2444	28465	165	183	112	1520	12535	4781
QF_BV/bench_ab	1109129	3385658	2783503	7797	10677	10143	11	11	10
QF_BV/sage/app2	218827	213889	217598	162	159	180	861	289	433
QF_BV/sage/app9	1137172	1984923	1495072	1301	1616	1595	466	464	465
QF_BV/sage/app8	389719	543033	260763	202	375	317	1515	1523	1506
QF_BV/sage/app5	1146022	1397666	1008205	1351	1638	1657	391	205	261
QF_BV/sage/app1	243452	509655	527177	1171	2421	2362	281	294	267
QF_BV/sage/app12	470245	847513	334077	1168	1322	844	313	298	201



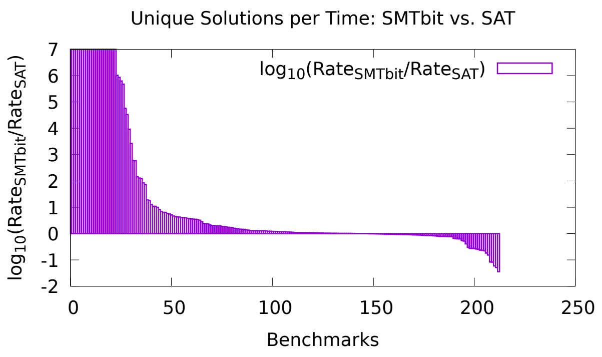
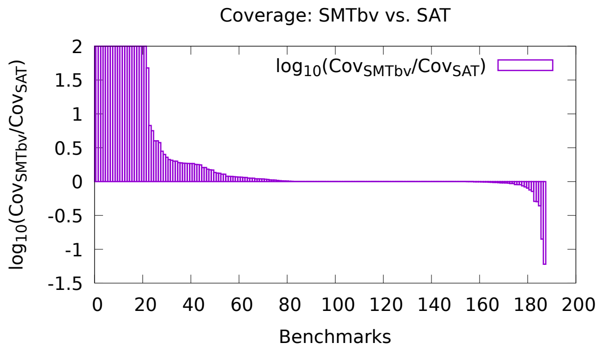


Figure 5.7: Speed comparison between the different approaches.

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,



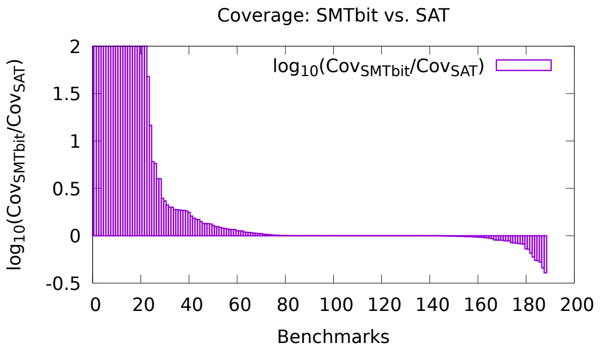


Figure 5.8: Coverage comparison between the different approaches.

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 GuidedSampler as an open-source tool[6] in C++, using Z3 [26] as the constraint solver. Z3 has native support for the MAX-SMT queries [9] that are required by

[6]The source code is available at `https://github.com/RafaelTupynamba/GuidedSampler`.

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 ϕ as an abstract syntax tree (AST), we look at the internal nodes of type bit-vector or Boolean. For example, in the formula $\phi = \phi_1 \vee \phi_2$, two of the internal nodes are the disjuncts ϕ_1 and ϕ_2. We sample solutions to ϕ 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 $\phi = \phi_1 \vee \phi_2$, we could pick ϕ_1 and ϕ_2 to be two of the coverage predicates, and choose additional predicates as subparts of ϕ_1 and ϕ_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 $\mathit{Vars}[\phi]$. 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 $\mathit{Vars}[\phi]$. 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 σ_0 to ϕ and then adding a new blocking clause $(\psi_1 \neq \psi_1[\sigma_0] \vee \psi_2 \neq \psi_2[\sigma_0] \vee \cdots \vee \psi_n \neq \psi_n[\sigma_0])$ to the formula that guarantees that future solutions will not come from the coverage class $G[\sigma_0]$. We then compute a new solution σ_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 $b = (b_1, b_2, \dots, b_n)$ 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 $\phi \wedge (\psi_1 = b_1) \wedge (\psi_2 = b_2) \wedge \cdots \wedge (\psi_n = b_n)$, 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 $b = (b_1, b_2, \dots, b_n)$ that define a coverage class and trying to find the solution closest to that class. We use the query MAX-SMT($\{\phi\}, S_b$), where the predicates are added as soft constraints $S_b = \{\psi_1 = b_1, \psi_2 = b_2, \dots, \psi_n = b_n\}$. Then repeat the process for different random vectors b. Each MAX-SMT call is guaranteed to eventually find a solution if ϕ is satisfiable, but the optimization problems can be expensive.

S0: SMTSampler. This baseline approach consists of applying SMTSampler to sample solutions to formula ϕ. 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 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.

Benchmarks	n	nodes	$\|Array\|$	$\|BV\|$	$\|Bool\|$	bits	preds
QF_AUFBV/ecc	4	179	0	27	7	1931	12
QF_ABV/bmc-arrays	3	855	1	1	0	53	82
QF_ABV/stp_samples	15	1139	1	24	0	192	58
QF_ABV/dwp_formulas	5	613	3	32	0	428	38
QF_ABV/egt	15	90	1	0	0	0	5
QF_ABV/bench_ab	15	314	1	0	0	2	3
QF_ABV/platania/...member	12	595	10	89	0	2856	2
QF_BV/bmc-bv	10	256	0	4	0	134	6
QF_BV/bmc-bv-svcomp14	8	7518	0	205	1055	7607	223
QF_BV/spear/zebra_v0.95a	9	571	0	185	0	2012	21
QF_BV/RWS	9	1086	0	21	0	3628	95
QF_BV/gulwani-pldi08	5	1146	0	130	0	950	185
QF_BV/stp_samples	14	793	0	22	0	200	10
QF_BV/brummayerbiere2	3	632	0	2	0	149	27
QF_BV/tacas07	3	8812	0	345	588	16620	91
QF_BV/bench_ab	13	21	0	1	0	24	1
QF_BV/sage/app2	13	231	0	24	0	206	7
QF_BV/sage/app9	8	271	0	35	0	391	17
QF_BV/sage/app8	15	978	0	93	0	1047	31
QF_BV/sage/app5	12	268	0	29	0	354	3
QF_BV/sage/app1	10	115	0	20	0	268	5
QF_BV/sage/app12	12	247	0	31	0	358	5

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).

Benchmarks	BC	BH	BS	S0	S1	S2	S3
QF_AUFBV/ecc	1829	188	854	6579	826	969	11377
QF_ABV/bmc-arrays	1263	44	660	1337	19003	19374	19122
QF_ABV/stp_samples	39750	9831	15082	11360	97511	679380	744289
QF_ABV/dwp_formulas	8162	7028	5688	404	2641	4227	31735
QF_ABV/egt	16276	8656	30984	16432	15561	49864	480017
QF_ABV/bench_ab	2254	151	3406	3118	2765	2530	6000
QF_ABV/platania/no_init_multi_member	1344	15	0	399	83	84	235161
QF_BV/bmc-bv	724	19	122	437	201	200	208
QF_BV/bmc-bv-svcomp14	4352	178	2866	2004	16363	19503	23861
QF_BV/spear/zebra_v0.95a	3896	50	2797	618	8135	9622	7384
QF_BV/RWS	2	0	0	187	84	80	53
QF_BV/gulwani-pldi08	4991	130	2524	9605	499158	566758	622999
QF_BV/stp_samples	34669	18015	81052	5387	431305	908703	1068203
QF_BV/brummayerbiere2	5	0	5	7	9	9	10
QF_BV/tacas07	5	0	0	53	51	58	78
QF_BV/bench_ab	158	2	134	99	153	149	152
QF_BV/sage/app2	12639	19501	20581	1583	769464	1004453	1035877
QF_BV/sage/app9	4910	1940	8829	13025	180458	257994	318534
QF_BV/sage/app8	14451	1312	20627	14635	209286	269363	328543
QF_BV/sage/app5	15344	1577	23736	6428	300547	437180	628252
QF_BV/sage/app1	6307	346	5731	3458	19765	20768	21473
QF_BV/sage/app12	8650	4371	9065	11766	330629	608030	683599

(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 $N_{\text{S3}}/N_{\text{X}}$ 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 GuidedSampler 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 SMTSampler 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

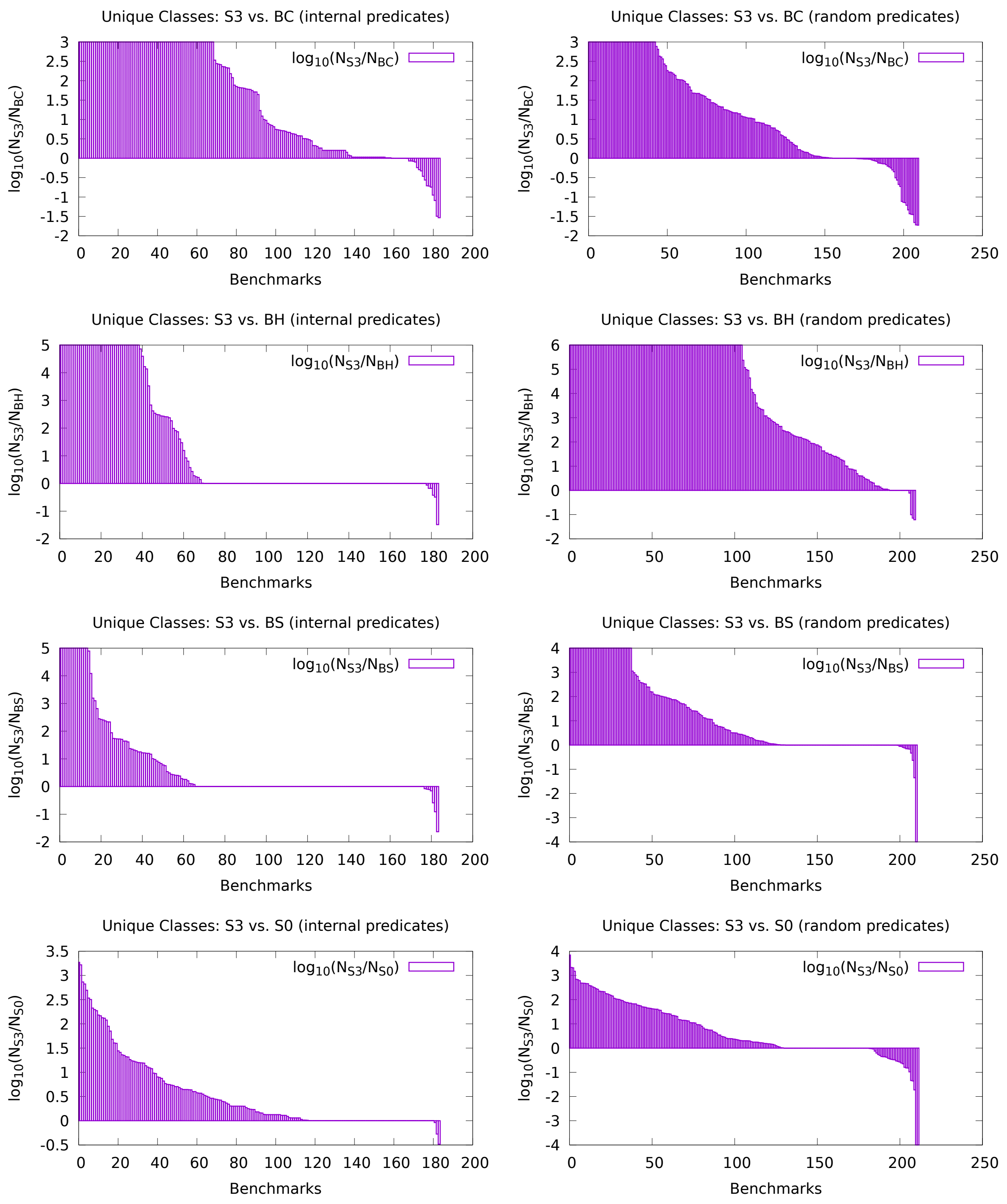


Figure 5.9: Unique classes covered: GUIDEDSAMPLER vs. baselines.

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.

Expression	Internal Predicates	Random Predicates
	mean [*low* − *high*]	*mean* [*low* − *high*]
$N_{\text{S3}}/N_{\text{BC}}$	3.4 [0.45 − 26]	4.2 [0.44 − 40]
$N_{\text{S3}}/N_{\text{BH}}$	2.6 [0.23 − 28]	38 [1.5 − 1018]
$N_{\text{S3}}/N_{\text{BS}}$	2.5 [0.34 − 19]	4.1 [0.52 − 31]
$N_{\text{S3}}/N_{\text{S0}}$	3.4 [0.60 − 19]	5.4 [0.49 − 60]
$N_{\text{S3}}/N_{\text{S1}}$	1.1 [0.56 − 2.2]	1.7 [0.54 − 5.2]
$N_{\text{S3}}/N_{\text{S2}}$	1.08 [0.64 − 1.8]	1.3 [0.54 − 3.2]
H_{BC}	1.0 [1.0 − 1.0]	1.0 [1.0 − 1.0]
H_{BH}	1.3 [0.93 − 1.8]	1.2 [0.85 − 1.7]
H_{BS}	1.3 [0.77 − 2.1]	2.2 [1.2 − 3.9]
H_{S0}	14 [1.5 − 129]	19 [2.8 − 130]
H_{S1}	2.7 [0.70 − 10]	8.5 [1.7 − 42]
H_{S2}	1.4 [0.86 − 2.2]	3.1 [1.4 − 6.9]
H_{S3}	1.3 [0.83 − 2.1]	2.9 [1.3 − 6.2]

solutions from the same few classes.

Table 5.11 shows mean values for the ratios $N_{\text{S3}}/N_{\text{X}}$, in the format 10^{μ} $[10^{\mu-\sigma} - 10^{\mu+\sigma}]$, where μ and σ are the average and standard deviation of $\log_{10}(N_{\text{S3}}/N_{\text{X}})$ 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

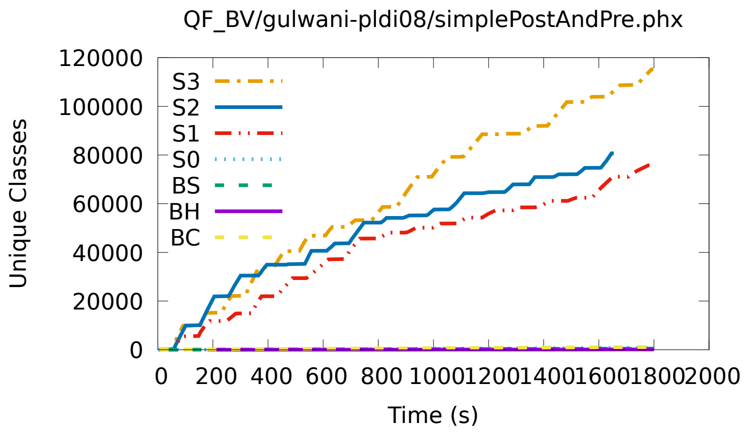


Figure 5.10: Unique classes over time for one benchmark (with random predicates).

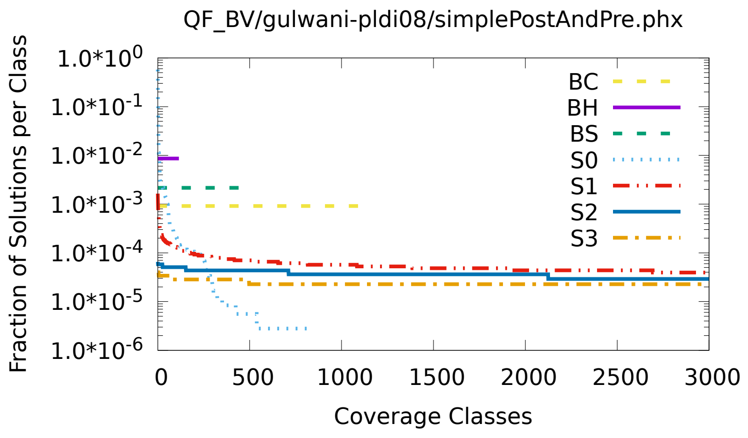


Figure 5.11: Uniformity of solutions over coverage classes for one benchmark (with random predicates).

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 $H_{\text{BC}} = 1$, 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 $H_{\text{S3}} \approx 1.3$ for internal predicates and $H_{\text{S3}} \approx 2.9$ 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 [50]. 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 [74, 31, 51].

One class of sampling methods is based on Markov Chain Monte Carlo (MCMC) algorithms [44, 45]. 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 [73, 44]. For example, [73] 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 [53]. 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 [31] 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 [30, 51] 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 [35]. UniGen [20] and UniGen2 [17] 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 [50]. 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.

For example, Markov Chain Monte Carlo (MCMC) methods [44, 45] could be adapted to work over SMT constraints. MCMC is the basis of most constrained-random verification techniques [45, 55, 77, 44]. 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 [44]. 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 that can also be adapted to the SMT domain is using a constraint solver to produce each sample. The internal search heuristics of the solver can be modified to generate more diverse samples [53]. 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, one could also consider adapting universal hashing techniques, such as UniGen [20] and UniGen2 [17], to work over SMT formulas. Those techniques 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 [18] is an approximate model counter for SMT formulas. It is applicable only to formulas in the bit-vector theory and works similarly to UniGen [20] and UniGen2 [17], 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 [50]. For example, techniques may be based on model counting [31], Markov Chain Monte Carlo (MCMC) [73, 45, 44], SAT solver search heuristics [53], combination of mutations [29, 28] and universal hashing [30, 51]. However, most sampling techniques have a general goal about the desired distribution of solutions, such as uniform distribution [20, 17], 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 [16], such as WAPS [36], 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

This final chapter presents the summary of our contributions, as well as several ideas for future work.

7.1 Summary of Contributions

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 [61, 60].

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 $2.5\times$ to $38\times$ 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.

7.2 Future Work

Our novel idea of computing atomic mutations and combining them to efficiently generate new samples has shown 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 [43, 23] and dynamic symbolic execution techniques [33, 66, 15, 14, 22, 48, 72, 3, 59, 2, 42, 64, 4, 62, 65, 34, 70, 5] are well-established techniques used in the testing of software, but which have also been applied to the testing of firmware [21] 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 [75, 76, 10, 38, 39, 57, 32]. 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 [41]. 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 [15] symbolic execution engine and AFL [76] fuzz testing tool.

Constrained-Random Verification (CRV). For hardware testing, *constrained-random verification* (CRV) [55] is heavily used in industry 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 [77, 54]. 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 [24], 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 [68] 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 [26] 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* [52], 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 [46] 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 [47] 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

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-guided synthesis". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE. 2013, pp. 1–8.

[2] Saswat Anand and Mary Jean Harrold. "Heap cloning: Enabling dynamic symbolic execution of java programs". In: *ASE*. 2011, pp. 33–42.

[3] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. "JPF-SE: a symbolic execution extension to Java PathFinder". In: *TACAS'07*. 2007.

[4] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. "Finding bugs in dynamic web applications". In: *ISSTA'08*. 2008.

[5] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 1083–1094. ISBN: 978-1-4503-2756-5.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a scala embedded language". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 1216–1225.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[8] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Hans van Maaren, and Toby Walsh. Vol. 4. IOS Press, 2009. Chap. 8.

[9] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ-An Optimizing SMT Solver." In: *TACAS*. Vol. 15. 2015, pp. 194–199.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 1032–1043.

[11] Robert Brummayer and Armin Biere. "Boolector: An efficient SMT solver for bit-vectors and arrays". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 174–177.

[12] Randal E Bryant. "Symbolic simulation—techniques and applications". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. ACM. 1991, pp. 517–521.

[13] Randal E Bryant, Shuvendu K Lahiri, and Sanjit A Seshia. "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions". In: *International Conference on Computer Aided Verification*. Springer. 2002, pp. 78–92.

[14] Jacob Burnim and Koushik Sen. "Heuristics for Scalable Dynamic Test Generation". In: *ASE'08*. Sept. 2008.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *OSDI'08*. Dec. 2008.

[16] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*. July 2014.

[17] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. "On Parallel Scalable Uniform SAT Witness Generation." In: *TACAS*. 2015, pp. 304–319.

[18] Supratik Chakraborty, Kuldeep S Meel, Rakesh Mistry, and Moshe Y Vardi. "Approximate Probabilistic Inference via Word-Level Counting." In: *AAAI*. Vol. 16. 2016, pp. 3218–3224.

[19] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. "A scalable approximate model counter". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, pp. 200–216.

[20] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. "Balancing scalability and uniformity in SAT witness generator". In: *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE. 2014, pp. 1–6.

[21] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. "CRETE: A Versatile Binary-Level Concolic Testing Framework". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham. 2018, pp. 281–298.

[22] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications". In: *ACM Trans. Comput. Syst.* 30.1 (2012), p. 2.

[23] Lori A. Clarke. “A program testing system”. In: *Proc. of the 1976 annual conference.* 1976, pp. 488–491.

[24] Henry M Cook, Andrew S Waterman, and Yunsup Lee. “TileLink cache coherence protocol implementation”. In: *White Paper* (2015).

[25] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[26] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 337–340.

[27] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. “GuidedSampler: Coverage-guided Sampling of SMT Solutions”. In: *Formal Methods in Computer-Aided Design (FMCAD), 2019.* IEEE. 2019.

[28] Rafael Dutra, Jonathan Bachrach, and Koushik Sen. “SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE. 2018.

[29] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. “Efficient Sampling of SAT Solutions for Testing”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE. 2018, pp. 549–559.

[30] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. “Embed and project: Discrete sampling with universal hashing”. In: *Advances in Neural Information Processing Systems.* 2013, pp. 2085–2093.

[31] Stefano Ermon, Carla P Gomes, and Bart Selman. “Uniform solution sampling using a constraint solver as an oracle”. In: *Conference on Uncertainty in Artificial Intelligence* (2012).

[32] Gordon Fraser and Andrea Arcuri. “EvoSuite: automatic test suite generation for object-oriented software”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6.

[33] P. Godefroid, N. Klarlund, and K. Sen. “DART: Directed Automated Random Testing”. In: *PLDI’05.* June 2005.

[34] P. Godefroid, M.Y. Levin, and D. Molnar. “Automated Whitebox Fuzz Testing”. In: *NDSS’08.* Feb. 2008.

[35] Carla P Gomes, Ashish Sabharwal, and Bart Selman. “Near-uniform sampling of combinatorial spaces using XOR constraints”. In: *Advances In Neural Information Processing Systems.* 2007, pp. 481–488.

[36] Rahul Gupta, Shubham Sharma, Subhajit Roy, and Kuldeep S. Meel. “WAPS: Weighted and Projected Sampling”. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* Apr. 2019.

[37] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. "A tale of two solvers: Eager and lazy approaches to bit-vectors". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 680–695.

[38] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. Bellevue, WA: USENIX Association, 2012, pp. 38–38.

[39] Allen D. Householder and Jonathan M. Foote. *Probability-Based Parameter Selection for Black-Box Fuzz Testing*. Tech. rep. Carnegie Mellon University Software Engineering Institute, Aug. 2012.

[40] Alexander Ivrii, Sharad Malik, Kuldeep S Meel, and Moshe Y Vardi. "On computing minimal independent support and its applications to sampling and counting". In: *Constraints* 21.1 (2016), pp. 41–58.

[41] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press. 2017, pp. 209–216.

[42] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. "jFuzz: A Concolic Whitebox Fuzzer for Java". In: *In NFM'09*. Apr. 2009.

[43] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19 (7 July 1976), pp. 385–394. ISSN: 0001-0782.

[44] Nathan Boyd Kitchen. *Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation*. University of California, Berkeley, 2010.

[45] Nathan Kitchen and Andreas Kuehlmann. "Stimulus generation for constrained random simulation". In: *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*. IEEE. 2007, pp. 258–265.

[46] Jakub Kuderski. *SMT-JIT: A toy Just-In-Time Compiler for evaluating SMT formulas (QF_AUFBV)*. `https://github.com/kuhar/smt-jit`. Accessed November 13, 2019.

[47] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *CGO'04*. Mar. 2004.

[48] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs". In: *CAV*. 2011, pp. 609–615.

[49] Sharad Malik and Lintao Zhang. "Boolean satisfiability from theoretical hardness to practical success". In: *Communications of the ACM* 52.8 (2009), pp. 76–82.

[50] Kuldeep S Meel. "Constrained Counting and Sampling: Bridging the Gap between Theory and Practice". PhD thesis. Rice University, 2017.

[51] Kuldeep S Meel, Moshe Y Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. “Constrained Sampling and Counting: Universal Hashing Meets SAT Solving.” In: *AAAI Workshop: Beyond NP*. 2016.

[52] Alexander Nadel. “Anytime Weighted MaxSAT with Improved Polarity Selection and Bit-Vector Optimization”. In: *Formal Methods in Computer-Aided Design (FMCAD), 2019*. IEEE. 2019.

[53] Alexander Nadel. “Generating Diverse Solutions in SAT.” In: *SAT*. Springer. 2011, pp. 287–301.

[54] Reuven Naveh and Amit Metodi. “Beyond feasibility: CP usage in constrained-random functional hardware verification”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, pp. 823–831.

[55] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. “Constraint-based random stimuli generation for hardware verification”. In: *AI magazine* 28.3 (2007), p. 13.

[56] Robert Nieuwenhuis and Albert Oliveras. “On SAT modulo theories and optimization problems”. In: *International conference on theory and applications of satisfiability testing*. Springer. 2006, pp. 156–169.

[57] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. “Feedback-directed random test generation”. In: *ICSE’07, Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA, May 2007, pp. 75–84.

[58] O Padon, KL McMillan, A Panda, M Sagiv, and S Shoham. “Ivy: interactive verification of parameterized systems via effectively propositional reasoning”. In: *PLDI. ACM* (2016).

[59] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. “Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software”. In: *ISSTA’08*. July 2008.

[60] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. “Uniform sampling of sat solutions for configurable systems: Are we there yet?” In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 240–251.

[61] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. “Bug synthesis: Challenging bug-finding tools with deep faults”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 224–234.

[62] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. “A Symbolic Execution Framework for JavaScript”. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP ’10. IEEE, 2010, pp. 513–528.

[63] Roberto Sebastiani. "Lazy satisfiability modulo theories". In: *Journal on Satisfiability, Boolean Modeling and Computation* 3 (2007), pp. 141–224.

[64] Koushik Sen and Gul Agha. "CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools". In: *CAV'06*. 2006.

[65] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript". In: *ES-EC/FSE'13*. To appear. Aug. 2013.

[66] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *ESEC/FSE'05*. Sept. 2005.

[67] Sanjit A. Seshia. "Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification". PhD thesis. Carnegie Mellon University, May 2005.

[68] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. "Programming by sketching for bit-streaming programs". In: *ACM SIGPLAN Notices*. Vol. 40. 6. ACM. 2005, pp. 281–294.

[69] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial sketching for finite programs". In: *ACM Sigplan Notices* 41.11 (2006), pp. 404–415.

[70] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *ICISS'08*. Dec. 2008.

[71] Marc Thurley. "sharpSAT-counting models with advanced component caching and implicit BCP". In: *SAT* 4121 (2006), pp. 424–429.

[72] Nikolai Tillmann and Jonathan de Halleux. "Pex - White Box Test Generation for .NET". In: *TAP'08*. Apr. 2008.

[73] Wei Wei, Jordan Erenrich, and Bart Selman. "Towards efficient sampling: Exploiting random walk strategies". In: *AAAI*. Vol. 4. 2004, pp. 670–676.

[74] Wei Wei and Bart Selman. "A new approach to model counting". In: *SAT*. Springer. 2005, pp. 324–339.

[75] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8.

[76] Michał Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl`. Accessed October 1, 2016.

[77] Yanni Zhao, Jinian Bian, Shujun Deng, and Zhiqiu Kong. "Random stimulus generation with self-tuning". In: *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009. 13th International Conference on*. IEEE. 2009, pp. 62–65.

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-Dīn al-Ṭūsī).

[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).

Rafael Tupynambá Dutra	University of California, Berkeley	2019
Koushik Sen	University of Illinois at Urbana-Champaign	2006
Gul Agha	University of Michigan	1985
John Henry Holland	University of Michigan	1959
Arthur Burks	University of Michigan	1941
Cooper Harold Langford	Harvard University	1924
Edwin Boring	Cornell University	1914
Edward B. Titchener	Universität Leipzig	1892
Eilhard Wiedemann	Universität Leipzig	1872
Karl Christian Bruhns	Friedrich-Wilhelms-Universität zu Berlin	1856
Johann Franz Encke	Universität Göttingen	1825
Carl Friedrich Gauss	Universität Helmstedt	1799
Johann Friedrich Pfaff	Universität Göttingen	1786
Abraham Gotthelf Kästner	Universität Leipzig	1739
Christian August Hausen	Universität Wittenberg	1713
Johann Andreas Planer	Universität Wittenberg	1686
Georg Pasch	Universität Wittenberg	1683
Michael Walther der Jüngere	Universität Wittenberg	1687
Johannes Andreas Quenstedt	Universität Wittenberg	1644
Christoph Notnagel	Universität Wittenberg	1630
Ambrosius Rhode	Universität Wittenberg	1610
Johannes Kepler	Tübinger Stift	1591
Tycho Brahe	Universität Wittenberg	1565
Caspar Peucer	Universität Wittenberg	1545
Georg Joachim Rheticus	Universität Wittenberg	1535
Nicolaus Copernicus	Università di Bologna	1499
Domenico Maria Novara da Ferrara	Università degli Studi di Firenze	1483
Regiomontanus	Universität Wien	1457
Basilios Bessarion	Mystras	1436
Gemistus Pletho	Mystras	1393
Demetrios Kydones	Thessaloniki	c.1365
Neilos Kabasilas	Thessaloniki	1363
Gregory Palamas	Constantinople	c.1336
Theodore Metochites	Constantinople	1315
Manuel Bryennios	Constantinople	c.1300
Gregory Choniades	Ilkhans Court at Tabriz	1296
Shams al-Dīn al-Bukhārī	Maragheh Observatory	c.1260
Nasir al-Din al-Tusi	Maragheh Observatory	c.1230
Kamal al-Din ibn Yunus	Persia	c.1200
Sharaf al-Dīn al-Ṭūsī	Persia	c.1175

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
Basilios 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-Din al-Tusi	Polymath, Architect, Philosopher, Physician, Scientist and Theologian
Kamal al-Din ibn Yunus	Mathematician, Chemist and Jurist
Sharaf al-Dīn al-Ṭūsī	Mathematician and Astronomer

Rafael Tupynambá Dutra	🇧🇷	University of California, Berkeley	🇺🇸	2019
Koushik Sen	🇮🇳	University of Illinois at Urbana-Champaign	🇺🇸	2006
Gul Agha	🇵🇰	University of Michigan	🇺🇸	1985
Carl Hewitt	🇺🇸	Massachusetts Institute of Technology	🇺🇸	1971
Seymour Papert	🟥	University of Cambridge	🇬🇧	1959
Frank Smithies	🇬🇧	University of Cambridge	🇬🇧	1937
G. H. Hardy	🇬🇧	University of Cambridge	🇬🇧	1903
E. T. Whittaker	🇬🇧	University of Cambridge	🇬🇧	1895
George Darwin	🇬🇧	University of Cambridge	🇬🇧	1871
Edward Routh	🟥	University of Cambridge	🇬🇧	1857
Isaac Todhunter	🇬🇧	University of Cambridge	🇬🇧	1848
William Hopkins	🇬🇧	University of Cambridge	🇬🇧	1830
Adam Sedgwick	🇬🇧	University of Cambridge	🇬🇧	1811
Thomas Jones	🇬🇧	University of Cambridge	🇬🇧	1782
Thomas Postlethwaite	🇬🇧	University of Cambridge	🇬🇧	1756
Stephen Whisson	🇬🇧	University of Cambridge	🇬🇧	1742
Walter Taylor	🏴󠁧󠁢󠁥󠁮󠁧󠁿	University of Cambridge	🇬🇧	1723
Robert Smith	🏴󠁧󠁢󠁥󠁮󠁧󠁿	University of Cambridge	🇬🇧	1715
Roger Cotes	🏴󠁧󠁢󠁥󠁮󠁧󠁿	University of Cambridge	🏴󠁧󠁢󠁥󠁮󠁧󠁿	1706
Isaac Newton	🏴󠁧󠁢󠁥󠁮󠁧󠁿	University of Cambridge	🏴󠁧󠁢󠁥󠁮󠁧󠁿	1668
Isaac Barrow	🏴󠁧󠁢󠁥󠁮󠁧󠁿	University of Cambridge	🏴󠁧󠁢󠁥󠁮󠁧󠁿	1652
Vincenzo Viviani	🟨	Università di Pisa	🟨	1642
Evangelista Torricelli	🟥🟨	Università degli Studi di Roma La Sapienza	🟥🟨	c.1641
Benedetto Castelli	🟥🟨	Università degli Studi di Padova	🟥	1610
Galileo Galilei	🟨	Università di Pisa	🟨	1585
Ostilio Ricci	🟥🟨	Università di Brescia	🟥🟨	c.1570
Niccolò Fontana Tartaglia	🟥	Università degli Studi di Padova	🟥	c.1516

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.