Cairo University
Faculty of Engineering
Department of Computer Engineering

FlashSolve

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by

Mostafa Ahmed Sobhy Ahmed

Supervised by

Prof. Amr Wassal

2023

Table of Contents

Introduction

Our project aims at enhancing the solution generation process for SystemVerilog (SV) constraints. The motivation behind this project is to develop an open-source tool that can be utilized by a wide range of users. SystemVerilog, a hardware description language (HDL), contains a sub-language for constraint definition and random variable definition, and SV runtimes contains an embedded solver for this constraint language. Our project aims to be such a solver, in stand-alone form that can be embedded in any SV runtime or toolchain.

By combining the power of ANTLR, the .NET runtime, and the state-of-the-art Z3 SMT solver, Our project offers a versatile and user-friendly tool for generating random solutions to SV constraints. The open-source nature of this project ensures accessibility, fostering further advancements in the field of hardware description languages and constraint solving techniques.

Our project is architected as an additional layer on top of Z3, implementing various innovative techniques for random sampling of SMT constraints. These techniques include a naive approach, Max-SMT, Universal Hashing, and a hybrid approach that combines these techniques. By incorporating these enhancements, we are able to sample random solutions by guiding the Z3 solver to random subsets of the input space

Module Objectives and Problem Definition

1. **Problem**: The problem we seek to address is the sampling problem encountered when generating random solutions using the Z3 solver. While the Z3 solver is highly effective in solving constraints, it lacks the inherent capability to produce truly random solutions.

2. **Objective**: we aim to enhance the solution generation process and provide users with a comprehensive tool that combines the power of the Z3 solver with effective sampling techniques.

3. **Approach**: Our approach involves implementing various techniques, such as Max Satisfiability (Max-SMT) and naive algorithms, to enable the generation of random solutions using the Z3 solver as a black box. By incorporating these techniques, we aim to enhance the solution generation process and address the sampling problem inherent in Z3.

4. **Outputs**: The output of our project is a collection of random solutions that satisfy the input constraints, generated through the combined use of the Z3 solver and our implemented sampling techniques. These solutions are presented in a structured format, providing users with a clear representation of the variable assignments that fulfill the specified constraints.

Literature Survey

What is Z3?

Z3 is an open source constraint solver and theorem prover developed at Microsoft Research. It is a powerful tool used for solving various logical and mathematical problems, including constraint satisfaction, model checking, and program verification. Z3 provides a high-level programming interface in multiple programming languages for constructing and manipulating logical formulas and solving them.

Z3 provides solvers that can check the satisfiability of logical formulas. Given a set of constraints, the solver determines if there exists a satisfying assignment of values to the variables that satisfies all the constraints. Z3 supports both satisfiability modulo theories (SMT) solving and quantifier-free solving.

Z3 employs highly optimized algorithms and heuristics to efficiently solve logical formulas. It utilizes various techniques, such as conflict-driven clause learning, theory-specific decision procedures, and efficient data structures, to improve performance.

What is the problem with Z3?

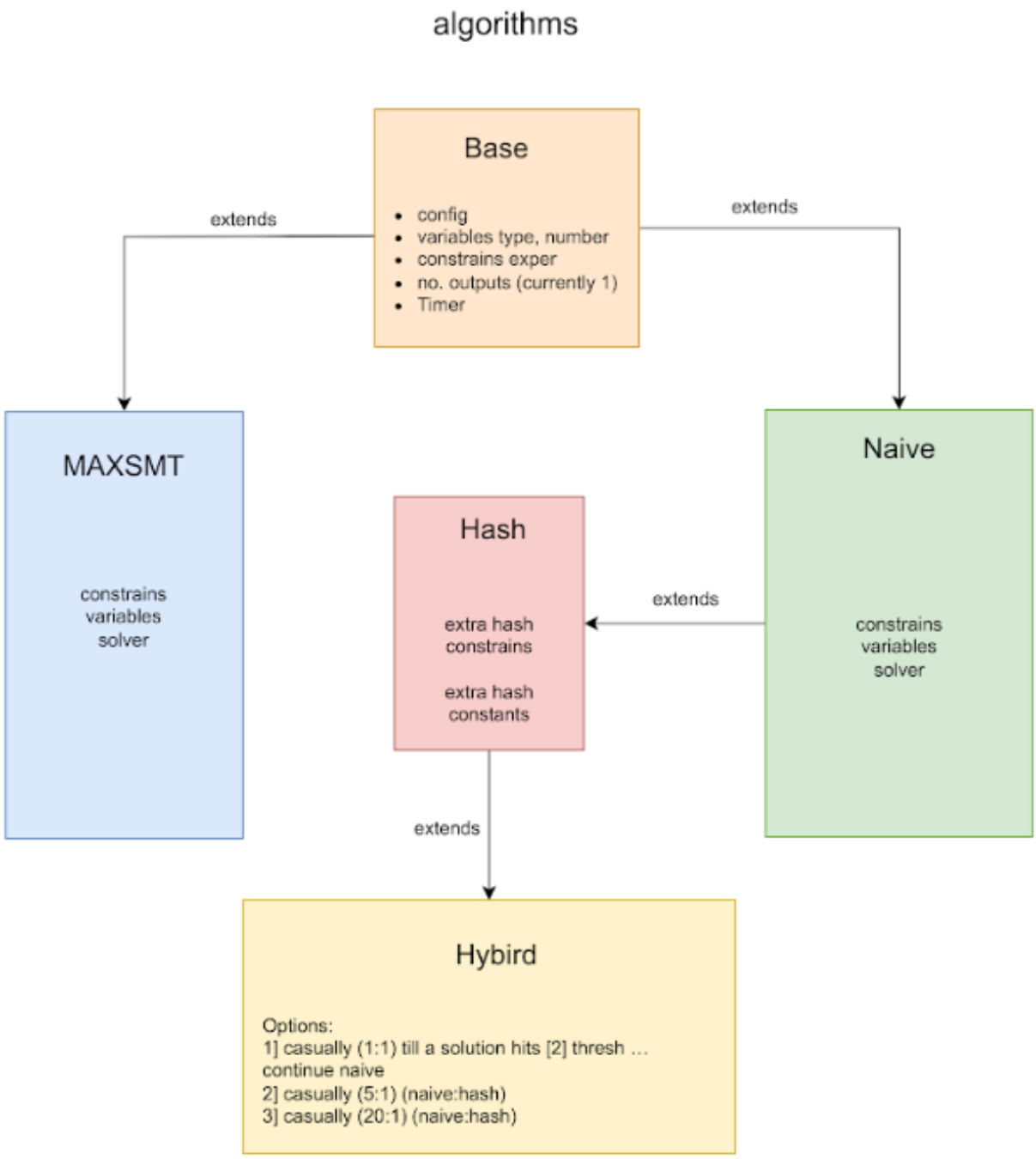
Z3, by default, provides deterministic behavior when solving constraints. This means that providing the same set of constraint formulae to Z3 will always produce the same solution given the same initial seed.

This behavior is intentional and helps ensure research integrity, reproducibility and consistency in the results. It allows for predictable debugging and testing of Z3 as well as the programs that depend on it..

Our tool’s sampler module has the responsibility of guiding Z3 towards random solutions of constraints in a purely black-box fashion. This is an open research problem (Efficient Sampling of SMT Constraints) in general but it can be approximated by several methods and algorithms.

System Architecture

this diagram shows the architecture of the code:



Modular Decomposition

Naive Approach

The naive approach involves simply exhausting the entire input space of the problem then choosing combinatorially between the generated solutions. This method is not suitable for huge input spaces, but it's ideal when :

1. **Solution Enumeration Is Required** : When a complete enumeration of all possible solutions to the constraints is already required (e.g. during the design of safety-critical systems), the naive approach is ideal as it's the only method that provably exhausts the entire solution space..

2. **Limited Solution Space**: If the problem has a relatively small number of solutions, the naive approach can be a practical choice, as it obtains all possible solutions with no need to rely on complex algorithms or heuristics.

3. **Satisfiability Analysis**: If the satisfiability of the problem's constraints is uncertain and it's not clear if any solutions exist at, the naive approach allows determining whether the problem is satisfiable as it doesn't involve augmenting the problem with any additional constraints.

4. **Reproducibility**: The naive approach reflects the same deterministic behavior of the underlying Z3 engine, meaning that given the same set of initial constraints, the exact sequence of solutions every time will be obtained on every run. This characteristic is very valuable for reproducibility, as it allows consistent analysis and comparison of the solutions obtained.

However, the naive approach is significantly non-competitive with more sophisticated algorithms and approaches due to

1. **Huge Computational Complexity**: As naive sampling is effectively a brute force search over the input space, it can be extremely computationally expensive or outright intractable for problems with dense solution spaces or mathematically complex (high dimensional) constraints.

2. **Performance Degradation**: As naive sampling involves repeatedly augmenting the problem instance with uniqueness-enforcing constraints, this can cause a sharp decrease in performance as the search over the very large space continues. As the solver finds more and more solutions, it's increasingly "burdened" by the need to avoid finding those same solutions again.

3. **Obliviousness To Problem Characteristics**: Some problems have a uniform structure that enables faster and more efficient solutions than brute force. For example, if the input space is dense (most points in the space are valid solutions), brute force

searching is extremely inefficient, but simple guessing-and-checking is both efficient and highly random.

Naive Algorithm

The algorithm steps can be described in pseudo code as follows :

1. **Initial Constraints**: The original user-defined constraints, those can come from anywhere, depending on the application that depends on FlashSolve. In our system, the initial constraints are the output of the SV2Z3 compiler, which transforms SystemVerilog (SV) into (2) Z3 formuli/ASTs.

2. **Checking Satisfiability**: Invoke Z3 on the constraints and see if they are satisfiable. Non-Satisfiability (UNSAT) means no assignment of values (meaning a model) can possibly satisfy all constraints. This marks the exit condition of the algorithm, as the valid solution space was entirely exhausted (or was zero-sized).

3. **Obtaining a Solution**: If, however, the result of step (2) was SAT (satisfiable), it means that Z3 has found a “model”, an assignment of constraint variables to valid domain values that results in all constraints evaluating to true.

4. **Adding Constraints to force Uniqueness of Solutions**: Since Z3 is inherently deterministic, running it on the same set of constraints will always produce the same solution. This can be worked around by adding a constraint to (or augmenting) the original problem to force Z3 to pick a different solution, this constraint is constructed from the solution found in step 3 and is basically logically equivalent to (NOT (&& (VAR1 == VAL1) (VAR2 == VAL2) …. (VARn == VALn))), effectively saying that any further solution has to satisfy that its values are not all equal to the previously found solutions, forcing uniqueness.

5. **Iterative Solution Generation**: Repeat steps 2-4 until the solver returns UNSAT.

So, The algorithm starts with the user-defined constraints and checks their satisfiability using Z3. If the constraints are satisfiable, a solution (model) is obtained. To ensure uniqueness of solutions, additional constraints are added based on the found solution. Steps 2-4 are iterated until the solver returns unsatisfiable, marking the end of the algorithm.

Roles of Naive Algorithm

The primary purpose of the naive algorithm within hybrid algorithms is not to obtain all solutions, but rather to assist other algorithms in achieving the total number of required solutions. The naive algorithm exhaustively explores the solution space, generating all possible solutions. However, its spread, or coverage of diverse solution options, is generally limited.

While the naive algorithm alone may not be ideal for obtaining comprehensive and diverse solutions, it plays a crucial supportive role alongside other algorithms that exhibit better spread characteristics. These other algorithms, which possess a higher capability for exploring different regions of the solution space, contribute to the generation of high-quality solutions.

By integrating the naive algorithm into the hybrid approach, it complements and assists the more effective algorithms in reaching the desired number of solutions. The exhaustive nature of the naive algorithm ensures that no solution is overlooked, while the other algorithms with better spread characteristics contribute to the diversity and quality of the generated solutions.

The naive algorithm plays a crucial role in hybrid algorithms, where it is often used in conjunction with a hashing algorithm. In hybrid algorithms, different techniques are combined to capitalize on their individual strengths and overcome their limitations.

For instance, in a hybrid algorithm with a 1:1 ratio, one solution is generated using the naive algorithm, while another solution is obtained using the hashing algorithm. Similarly, in a hybrid algorithm with a 1:5 ratio, one solution is derived from the hashing algorithm, and five solutions are generated using the naive algorithm. The ratio can vary depending on the specific requirements and considerations of the problem.

The naive algorithm is particularly valuable when the number of required solutions has not been achieved through other techniques such as hashing or more complex algorithms like maximum satisfiability (max SMT). If an algorithm like max SMT or hashing yields an unsatisfiable (UNSAT) result before reaching the desired number of solutions, the naive algorithm is employed to continue generating solutions until the required number is obtained.

By incorporating the naive algorithm into the hybrid approach, the algorithm ensures that the desired output count is met, even if other techniques fall short. This usage of the naive algorithm ensures that the required number of solutions is reached, enhancing the completeness and effectiveness of the overall hybrid algorithm.

The naive algorithm is an essential component of hybrid algorithms, often employed alongside hashing algorithms. It contributes to the solution generation process, either in combination with other techniques or as a fallback mechanism to achieve the desired number of outputs. By utilizing the strengths of both the naive algorithm and other techniques, hybrid algorithms offer a comprehensive and reliable approach to solving complex problems.

Naive Class

The "Naive" class extends the base class and incorporates a parametric constructor that accepts an object representing the configuration file, the number of outputs, and the problem to be solved.

Within the Naive class, the "runNaiveAlgorithm" function is implemented. This function executes the naive algorithm iteratively until it either achieves the required number of outputs or reaches a specified threshold. The purpose of this function is to generate solutions by applying the naive algorithm repeatedly until the desired output count or threshold is met.

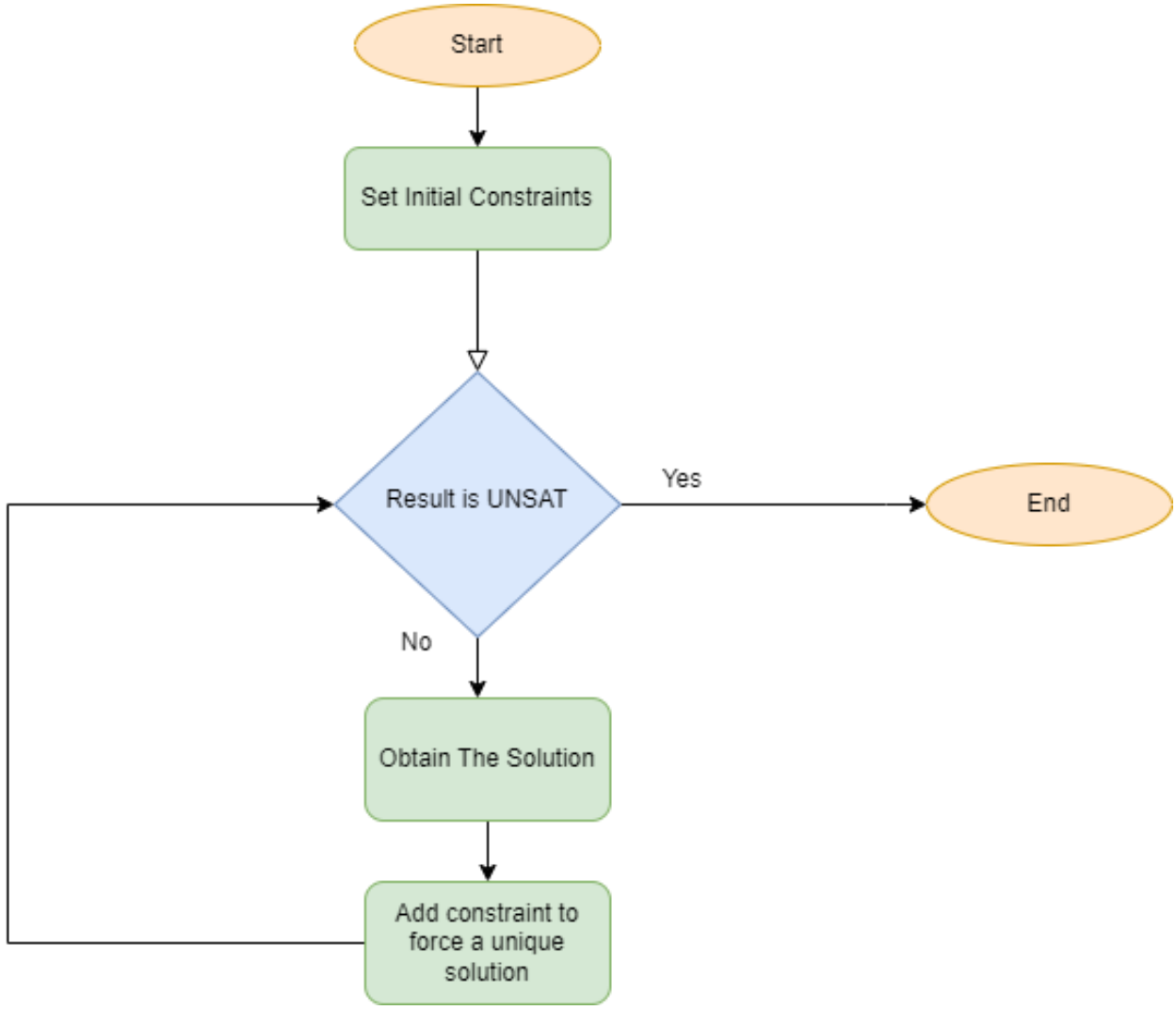
Additionally, the Naive class includes a "test" function, which serves the purpose of evaluating the performance of the naive algorithm in solving the given constraints. This evaluation involves executing the naive algorithm with a small sample size of outputs, typically 50, to assess both the time taken and the spread achieved. The intention is to compare the performance of the naive algorithm with that of other algorithms under consideration. By conducting this comparison, the test function aims to identify the most suitable candidate algorithm.

The test function applies the naive algorithm to solve the constraints using the small sample size of 50 outputs. It measures the time taken by the naive algorithm to obtain these outputs and evaluates the spread achieved, which represents the extent of coverage of the constraint space. By considering the time and spread, the test function

performs a comprehensive analysis and comparison of the naive algorithm against other potential algorithms.

The objective of the test function is to determine the algorithm that exhibits the most desirable characteristics based on the evaluation criteria. In particular, it aims to identify an algorithm with optimal time efficiency and a sufficiently high spread. This process enables the selection of the most suitable candidate algorithm from the pool of algorithms being considered.

The algorithm can also be described in Flowchart format as follows :



MAX-SMT Approach :

The Max-SMT depends on a generalization of the SMT problem called MAX-SMT.

MAX-SMT as a problem class :

MAX-SMT is a variant of the SMT problem where the constraints can be either “Soft” or “Hard”. Any model to a MAX-SMT constraint problem must satisfy all hard constraints and also satisfy the maximum possible number of soft constraints, but can possibly violate all of them. Yet another generalization of this is the weighted MAX-SMT problem, where every soft constraint is associated with a weight, and the goal - in addition to satisfying all hard constraints - is to maximize the total weight of all achieved soft constraints. Weighted MAX-SMT is the most general formulation of the SMT problem, it reduces naturally into unweighted MAX-SMT by setting all the weights to 1, and unweighted MAX-SMT reduces degenerates into traditional SMT by simply not asserting any soft constraints at all.

Intuitively, soft constraints enable us to set “soft goals” or preferences for the solver. Solutions that satisfy the preferences are better than those that don’t, but those that don’t are also acceptable as long as the hard constraints are not violated. This enables us to implement randomization by simply softly asserting that all variables have random values picked at random from their domains, and leaving the solver to discover values that are as much as possible identical to the (softly-asserted) random values stipulated by us, but can also deviate from them if the hard constraints require so.

The Naive implementation of MAX-SMT forcing :
Naively, then, a MAX-SMT solver could be to implement random sampling of SMT solutions naively as follows :

- First, set all user-defined constraints (those compiled from the user input) as the hard constraints for the solver
- Then, for each variable, pick a random value uniformly from its domain and, possibly, by taking into consideration unary constraints on the variable
- Then, construct a soft constraint for each variable asserting that it’s equal to the corresponding random value chosen in step 2
- Set the solver’s soft constraints to those constraints constructed in step 3
- Run the solver and record the solution

What we’re doing is, in simple words, telling the solver to “prefer” random values for each variable, but to not prefer it so much as to make it a hard requirement for a solution. This is because random values picked completely ignoring all non-unary constraints, it’s very unlikely that all random values are together a valid model for the system of constraints. Thus, we only allow the randomization assertions to “advise” the solver, they are soft

suggestions to guide the solver to random subspaces in the solution space, but not actually hard assertions.

The problem with this approach is that it's not satisfactorily efficient. Intuitively, if the advice is "bad", meaning if most subsets of randomization assertions are inconsistent as a model, then it's a burden on the solver that slows it down. The solver spends a lot of time being "misled" by the advice and finding out on its own that it's actually bad advice, then discarding it. In a more concrete sense, Z3 is not an optimizer, it has a single optimization algorithm which is not competitive with state-of-the-art optimizers.

Thus, a new, but preferably equivalent, algorithm is needed.

Profile-guided subset-and-randomize :

Consider the following algorithm :

- Find a random non-empty subset of the problem's variable
- Construct randomization assertions for those variables
- Run the solver and record the solution
- If the subset's randomization assertion is satisfied, increase the probability that the subset chosen has will be chosen again in the future
- Repeat 1-4 till the problem becomes unsatisfiable or the desired number of solutions is obtained

Intuitively, this is just a relaxation of the straightforward MAX-SMT randomization above. If the subset chosen happens to be the entire set of variables, the algorithm reduces to naive randomization.

However, this algorithm also leaves open the possibility of arbitrary subsets of variables being chosen for randomization. Intuitively, this is good because the less we "micromanage" the solver, the less we assert random values for variables, the more freedom the solver has and the less time it wastes on discarding bad advice. Moreover, the algorithm profiles its choices and remembers the subsets that result in successful randomizations, and gives them a higher chance of being chosen in the future.

As a bonus, the algorithm doesn't require a MAX-SMT optimizer, it can simply use plain hard constraints to assert randomization, using a special Z3 API assert "local constraints" that are valid only in the scope of a single check call. This is much more efficient than MAX-SMT solving.

Exploration Vs. Exploitation

There is a difficult question in the algorithm design : should subsets that result in more successful randomization be chosen more often ? Or should we try out new combinations of variables in order to find an even better and more successful combination than the one we have ?

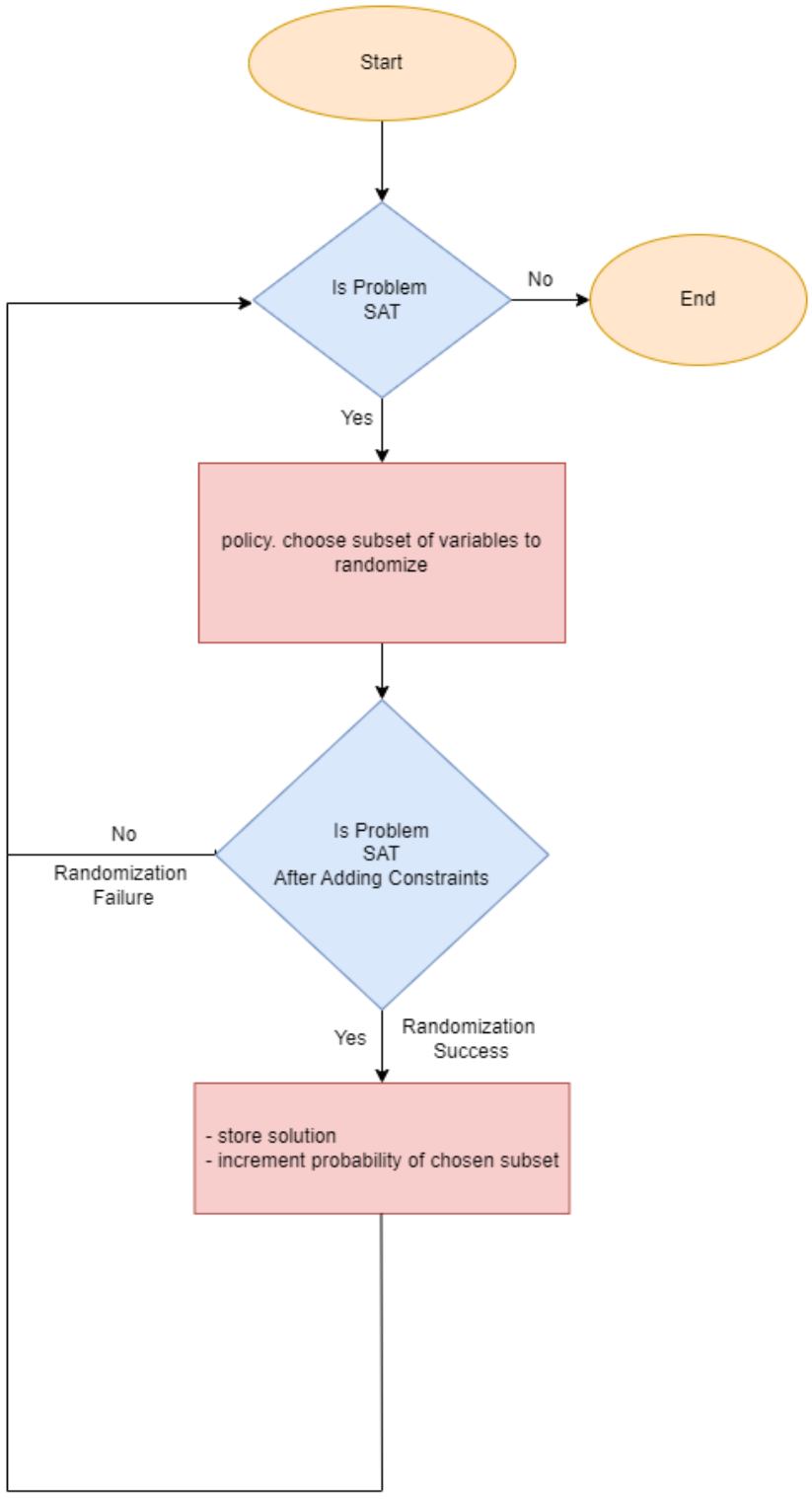
This is the oft-studied Exploration vs. Exploitation dilemma in Reinforcement learning. Exploitation acts greedily with respect to the current information. Here the information is the number of successful randomizations per variable subset, and acting greedily means picking the subset that maximizes the chance of a successful randomization (empirically estimated using counts). Exploration builds seeks new and more accurate information, and is as crucial as exploitation, especially at the beginning of the state space exploration.

A common Exploration-Exploitation balancing act is the Epsilon Greedy strategy. This strategy acts greedy with high probability 1-e, where e (epsilon) is a suitable small real number. Once in a while, the Epsilon Greedy strategy will with probability e choose to be more exploratorary, in order to ensure that it doesn't miss out on useful information. This strategy is mathematically optimal in a lot of ways, but it needs to be paired with a high initial epsilon in order to be truly successful in a lot of ways, the "Optimistic Initialization" allows the strategy to build crucial needed info at the beginning of the solution process.

Implementation : Randomizers and Policies
Thus, the implementation of the subset-and-randomized algorithm in practice is

- Check if the user defined constraints are satisfiable
- Consult a policy to see which subset of variables to randomize this iteration
- Consult the chosen variables' randomizers to get a random value for each one and construct a randomization assertions for each one
- Check satisfiability with the temporary addition of those randomization assertion to the solver's constraints
- If not satisfiable, go to 1
- Otherwise if satisfiable, record the solution to the model database and increment the number of successful randomization for the subset chosen
- Go to 1

The algorithm can also be described in Flowchart format as follows :


Start
Is Problem SAT
No
End
Yes
policy. choose subset of variables to randomize
Is Problem SAT After Adding Constraints
No
Randomization Failure
Yes
Randomization Success
- store solution
- increment probability of chosen subset

A “Policy” is the algorithm’s abstraction of all the machinery needed to act on the profiled information. Epsilon Greedy is one particular instance of a policy, but the algorithm can in principle leverage any policy, indeed it can even learn dynamic policies using Machine Learning, Reinforcement Learning, Evolutionary Programing, or other learning paradigms. Dynamically adapting itself to the data as it gathers more and more statistics about the problem.

Randomizers are the algorithm’s abstraction of all the machinery needed to pick a random value from a variable’s domain. A Blind Randomizer is an instance of a randomizer that takes nothing into account but the variable’s domain. One can imagine more sophisticated randomizers that take unary and perhaps even binary constraints into account, effectively doing some local and heuristic constraint solving of its own.

With the algorithm parameterized over both policies and randomizers (themselves parametrized and tunable through constructors and inheritance), the subset-and-randomized algorithms have a huge range of possible behavior. A future research direction might be to perform hyperparameter search over a space of policies and randomizers in order to find the most efficient time-randomness tradeoff in the found solutions.

Test Function

Upon parsing a System Verilog file containing SV constraints, the user is presented with two options for further processing. Firstly, they have the flexibility to explicitly specify a particular algorithm that will be employed for solving the constraints. This allows for a targeted and deterministic approach.

Alternatively, the user can choose to initiate the execution of a test function. The purpose of this test function is to evaluate and compare multiple algorithms concurrently, leveraging the power of parallel processing using different threads. This parallel execution enables efficient and simultaneous exploration of various algorithmic approaches.

During the test function's execution, each algorithm is subjected to rigorous analysis and evaluation based on predefined criteria. The user specifies an equation within the configuration file, which assigns weights to different factors such as spread and time. These factors represent crucial considerations in determining the effectiveness and efficiency of each algorithm.

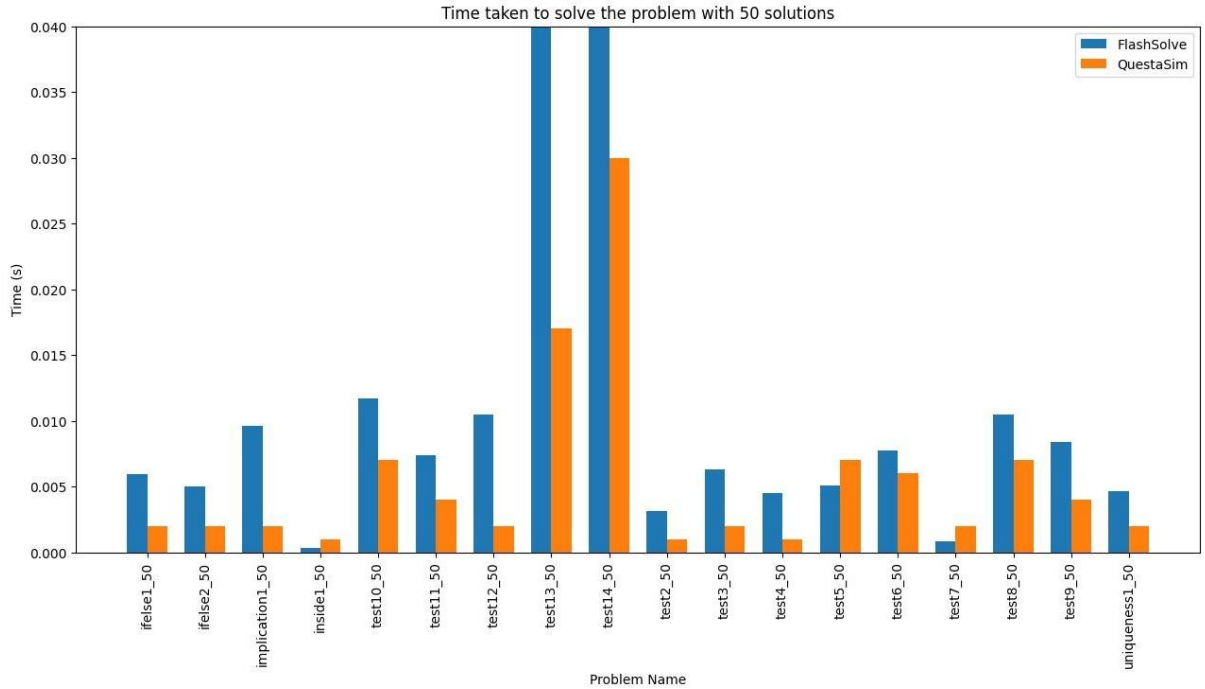
The evaluation process involves executing the algorithms with different inputs and measuring their performance in terms of time taken and spread achieved. The spread represents the ability of the algorithm to cover a wide range of constraints effectively.

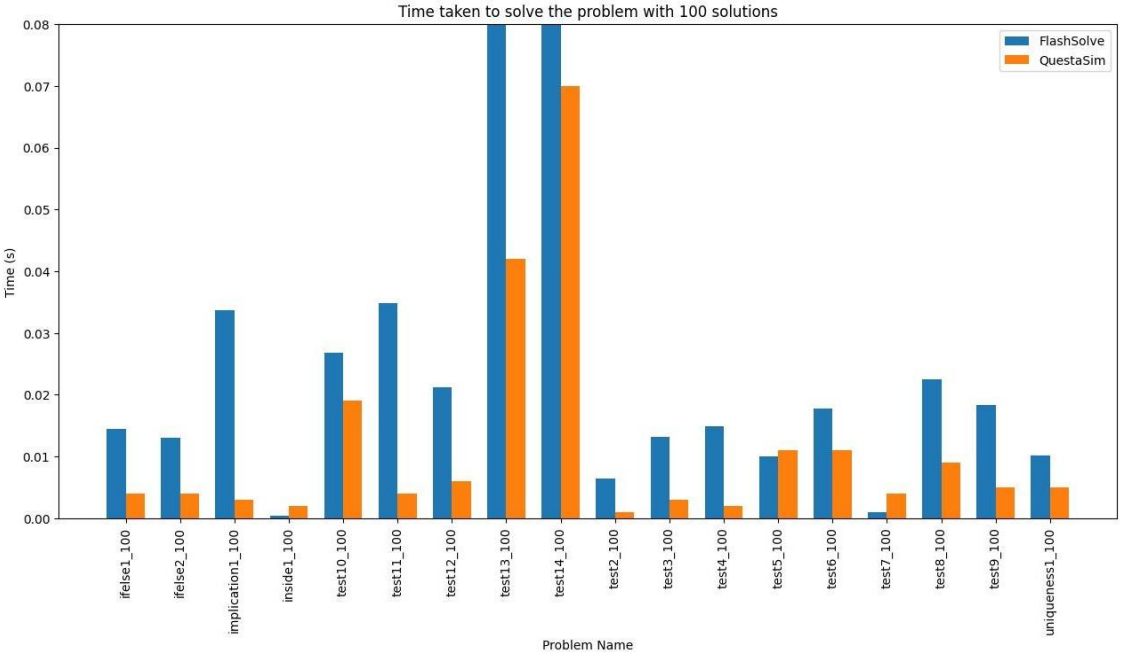
To determine the most suitable candidate algorithm, the test function assigns a score to each algorithm based on the equation specified in the configuration file. This equation combines the weights assigned to spread and time, providing a means to prioritize certain characteristics over others. Generally, the equation favors algorithms that achieve a higher spread while minimizing the time required for constraint solving.

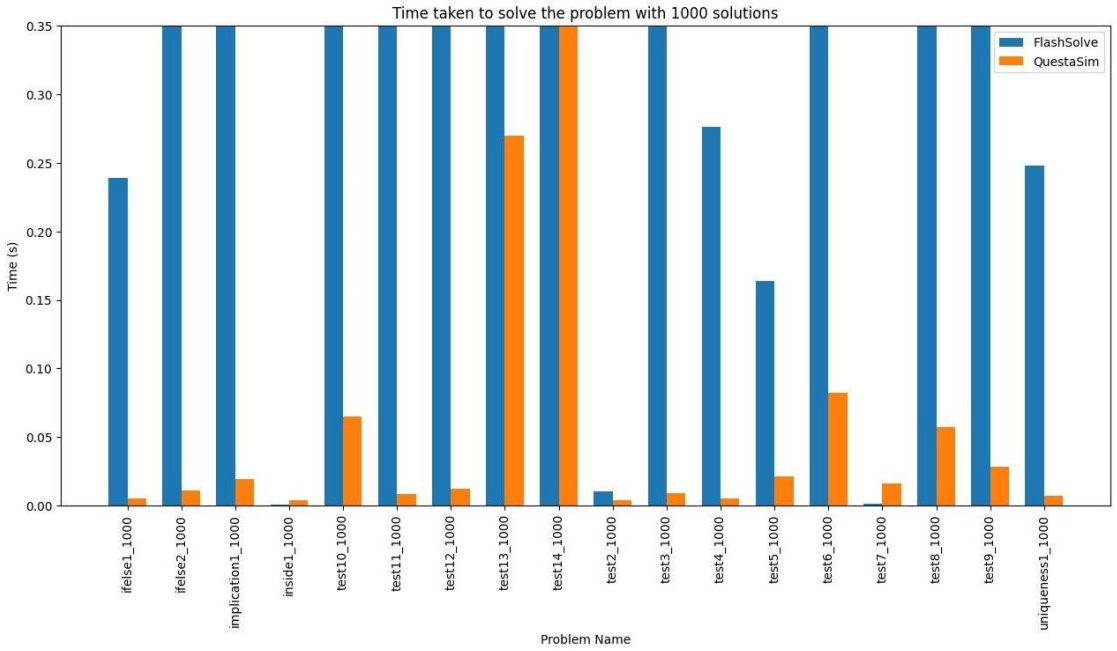
Ultimately, the test function selects the algorithm that attains the highest score as the candidate algorithm to be utilized for solving the given set of constraints. By considering the equation's weighted factors and evaluating the algorithms' performance on parallel threads, this approach enables an informed decision based on the desired trade-off between spread and time efficiency.

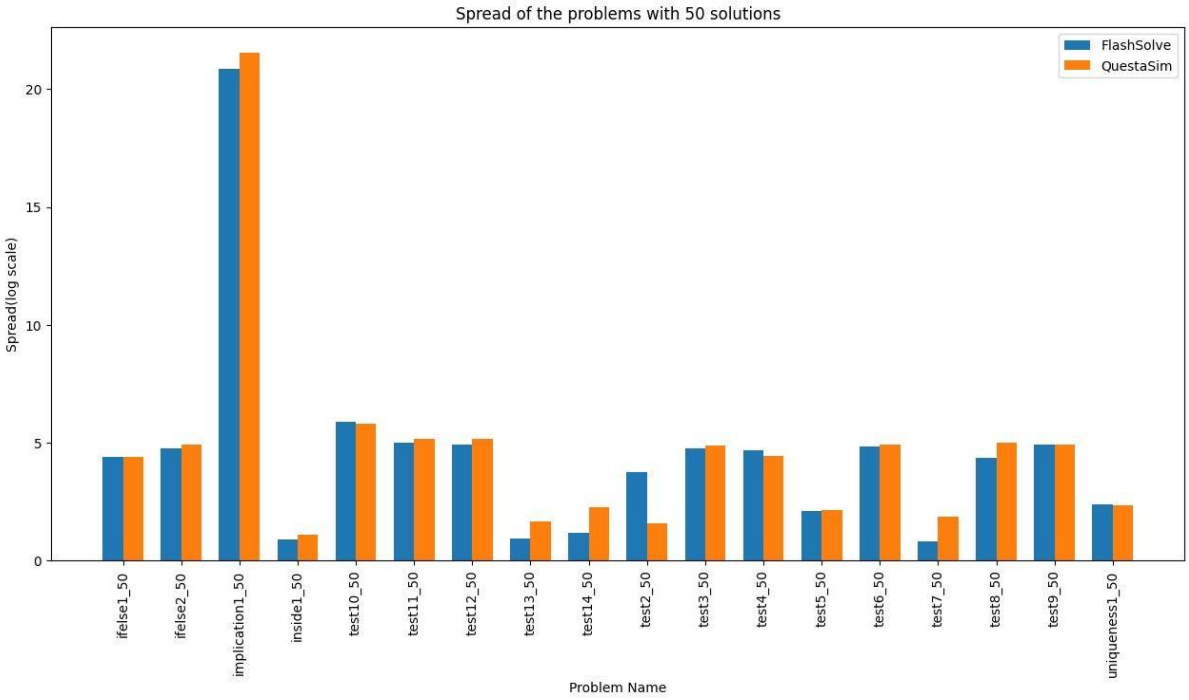
The test function conducts a comprehensive evaluation of multiple algorithms concurrently, assessing their performance based on spread and time. By employing an equation specified in the configuration file, the test function calculates scores for each algorithm, ultimately selecting the one with the highest score as the candidate algorithm for solving the provided constraints.

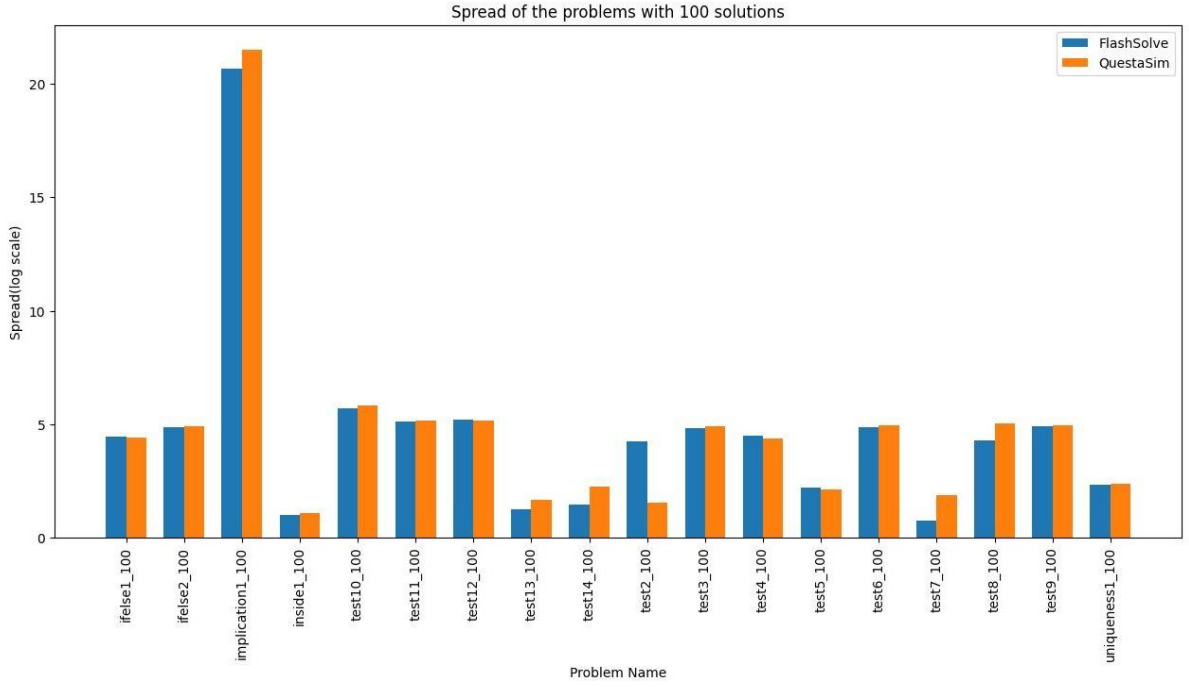
Module Testing

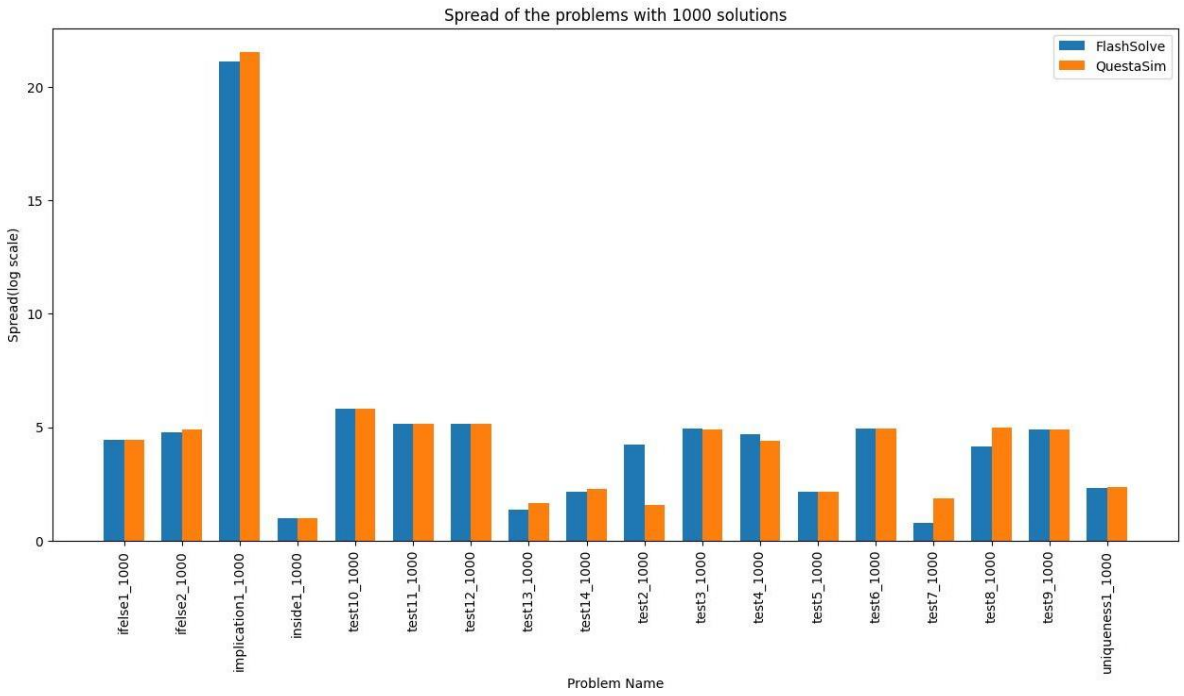



Time taken to solve the problem with 100 solutions
FlashSolve
QuestaSim
Time (s)
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
ifelse1_100
ifelse2_100
implication1_100
inside1_100
test10_100
test11_100
test12_100
test13_100
test14_100
test2_100
test3_100
test4_100
test5_100
test6_100
test7_100
test8_100
test9_100
uniqueness1_100
Problem Name


Time taken to solve the problem with 1000 solutions
FlashSolve
QuestaSim
Time (s)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
ifelse1_1000
ifelse2_1000
implication1_1000
inside1_1000
test10_1000
test11_1000
test12_1000
test13_1000
test14_1000
test2_1000
test3_1000
test4_1000
test5_1000
test6_1000
test7_1000
test8_1000
test9_1000
uniqueness1_1000
Problem Name


Spread of the problems with 50 solutions
FlashSolve
QuestaSim
Spread(log scale)
0
5
10
15
20
ifelse1_50
ifelse2_50
implication1_50
inside1_50
test10_50
test11_50
test12_50
test13_50
test14_50
test2_50
test3_50
test4_50
test5_50
test6_50
test7_50
test8_50
test9_50
uniqueness1_50
Problem Name


Spread of the problems with 100 solutions
FlashSolve
QuestaSim
Spread(log scale)
0
5
10
15
20
ifelse1_100
ifelse2_100
implication1_100
inside1_100
test10_100
test11_100
test12_100
test13_100
test14_100
test2_100
test3_100
test4_100
test5_100
test6_100
test7_100
test8_100
test9_100
uniqueness1_100
Problem Name



Upon comparing the results of the Max-SMT approach implemented in our project with QuestaSim, a tool developed by Siemens, we observed some interesting findings. In terms of execution time, we noticed that our project performs relatively well for small sample sizes, such as 50. However, as the sample size increases, the performance of our project falls short compared to QuestaSim.

On the other hand, when examining the spread of solutions, our project excels and demonstrates remarkable performance. In fact, our project is often on par with QuestaSim and, in some cases, even surpasses it in terms of solution spread. This indicates that our project has successfully implemented techniques that facilitate generating diverse and varied solutions to the given constraints.

While our project may not match the time efficiency of QuestaSim for larger sample sizes, it compensates by providing an impressive spread of solutions. This characteristic can be highly advantageous in scenarios where obtaining a wide range of solutions is crucial for thorough analysis, testing, and validation of system Verilog designs.

So, we can say our project strikes a balance between time efficiency and solution spread. It may not outperform QuestaSim in terms of execution time for larger samples, but it shines in generating diverse solutions that closely match or even surpass the spread achieved by QuestaSim. This feature positions our project as a valuable tool for users seeking comprehensive and varied solution sets for their system Verilog constraints.