An Efficient and Uniform CSP Solution Generator Generator

Ghiles Ziat
EPITA Reasearch Lab (LRE), Le Kremlin-Bicêtre, France

Martin Pépin
Université Caen Normandie, ENSICAEN, CNRS, Normandie Univ, GREYC UMR 6072, F-14000 Caen, France

Abstract

Constraint-based random testing is a powerful technique which aims at generating random test cases to verify functional properties of a program. Its objective is to determine whether a function satisfies a given property for every possible input. This approach requires firstly defining the property to satisfy, then secondly to provide a "generator of inputs" able to feed the program with the inputs generated. Besides, function inputs often need to satisfy certain constraints to ensure the function operates correctly, which makes the crafting of such a generator a hard task. In this paper, we are interested in the problem of manufacturing a uniform and efficient generator for the solutions of a CSP. In order to do that, we propose a specialized solving method that produces a well-suited representation for random sampling. Our solving method employs a dedicated propagation scheme based on the hypergraph representation of a CSP, and a custom split heuristic called *birdge-first* that emphasizes the interests of our propagation scheme. The generators we build are general enough to handle a wide range of use-cases. They are moreover uniform by construction, iterative and self-improving. We present a prototype built upon the *AbSolute* constraint solving library and demonstrate its performances on several realistic examples.

2012 ACM Subject Classification Computing methodologies → Randomized search

Keywords and phrases Constraint Programming, Property-based Testing

Digital Object Identifier 10.4230/LIPIcs.CP.2025.40

1 Introduction

The objective of this work is to propose a technique for building an efficient and uniform sampler of solutions from a Constraint Satisfaction Problem (CSP).

Generating uniformly distributed solutions to a Constraint Satisfaction Problem can be useful in various applications where fairness, diversity, unbiased testing, or comprehensive exploration of solution spaces is required. For resource allocation and fair division, applications like cloud computing and fair task distribution rely on uniform sampling to guarantee equity [27]. In machine learning, uniform solution generation helps creating diverse training datasets for constraint-based domains [1]. Cryptography benefits from uniform sampling for unpredictable key generation [6]. Also, applications like network testing leverage uniform sampling for unbiased evaluation of routing algorithms [12].

Also, crafting a uniform sampler is particularly useful in the context of Property Based Testing [3] (PBT), and more specifically in random testing in which they are refered to as *generators*. Random testing is a black-box testing technique where programs are tested by generating random, independent inputs, provided by a generator. Results of the output are compared against software specifications to verify that the test passes or fails. While generarating inputs uniformly at random can be straightforward, it is often necessary to generate inputs that satisfy certain number of prerequisites, while assuring a good coverage of the input space and maintaining reasonable execution time. Frameworks *à la* QuickCheck usually deal with this problem by providing *rejection sampler* combinators: a candidate input

is generated using a base generator and kept if it satisfies some constraints, or discarded otherwise. In the latter case, the test is not executed and the process is generally repeated a fixed number of time until a candidate is found or giving up. This approach is simple, both to use and to implement, and is uniform – given that the base generator is uniform itself. However, it might not be efficient in scenarios where the solution space is sparse or where constraints are complex. Constraint solving provides more powerful mechanisms to travel the solution space, making it a valuable improvement over repeated rejection sampling. Even though this improvement comes with the cost of importing the heavy machinery of a constraint solver, several works have shown that it makes the overall process more time and resource-efficient when the solution space is sparse[22, 26, 29, 2].

QuickCheck's rejection sampling provides a straightforward way to create a generator, albeit one that may be very slow. In contrast, constraint-solving techniques adopt a more computationally intensive process to construct a fast generator. Our proposed incremental approach seeks to strike a balance between these, aiming to achieve an optimal trade-off between the speed of generator construction and runtime efficiency. This balance is particularly beneficial in scenarios where there is little or no prior information about how many times the generator will be used. In such cases, the upfront cost of constructing a highly optimized generator through constraint solving may be unjustified if the generator is used only a handful of times. Conversely, relying solely on a quickly constructed, but slow generator, may result in suboptimal performance if the generator is reused extensively. By incrementally refining the generator as needed, our approach adapts to different usage patterns, providing a flexible and efficient solution regardless of the frequency of generator usage. This scalability is achieved through our novel selection structure, which is designed to handle more complex forms, such as non-fixed-size structures (e.g., list matrices), a challenge that many existing approaches fail to address. Instead of immediately generating examples, our method abstracts away the generation process, allowing for dynamic adjustment and refinement of the generator based on constraints. It thus incrementally *generates* a generator that encapsulates the desired properties and constraints, hence the title of this paper.

1.1 Contributions of the paper

- A general method to produce fast and self-improving uniform samplers of CSP solutions using a dedicated solving algorithm for random generation.
- A graph based split heuristic and propagation scheme, well fit to partition a problem into independent sub-problems.
- A heuristic Huffman-like representation for the solutions which minimizes the cost of uniform choices.
- A property based testing API, *à la quickcheck*, allowing for a transparent usage of our hybrid approach between rejection sampling and constraint based solving. The implementation is available at:
  `https://osf.io/u4r5q/files/osfstorage?view_only=84af8ee65a6c495d98cb7b7bfad8c54b`

1.2 Outline

This paper is organized as follows: Section 2 defines the concept of generators in general and gives some insight of what is expected from a *good* generator. Section 3 introduces the main mechanisms needed to build a constraint-based generators. Section 4 is the main contribution of this work: it addresses the problem of building a *propagation-exploration*

scheme well-suited for the design of a constraint based generator. Section 6 presents our implementation and show its performances on some benchmark. Finally, Section 7 presents the related works and Section 8 summarizes our work and discusses its future continuations.

2 What is a (Good) Generator?

CSP's search spaces are defined by variables to which are associated finite bound domains. A point in this search space in generally called an instance, but for our purpose we will call those samples.

▶ **Definition 1** (Sample). *Given a set of variables $\mathcal{V}$, and a set of domains $\mathcal{D}$ denoting the possible values of variables, a sample is a total mapping from variables to values. We note the set of samples $\mathcal{S} = \mathcal{V} \to \mathcal{D}$*

We distinguish two kinds of samples, the ones that satisfies all the constraint of a given CSP *i.e.* the solutions, and the ones that violate at least one constraint.

▶ **Definition 2** (Solution Generator). *Given a CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$, a solution generator is a function g that takes a random state and produces a solution $s \in S$, such that:*

$$\forall r, \forall c \in \mathcal{C}, \quad c(g(r)) \text{ holds}$$

where the first quantification is over all the possible random states r.

This definition ensures that every generated sample satisfies all constraints of the CSP. Moreover, the use of random states[1] is needed so that the generation process can be made reproducible by controlling or restoring the state. Good generators should be:

- **Correct:** They should respect the constraints they are subject to. For a data type representing positive integers, a generator should ensure that it only produces positive integers.
- **Uniform:** They should be able to thoroughly explore the input space. In other lines of work, diverseness is ensured by building surjective generators (*i.e.* every possible solution can be generated with *a non-zero probability* [9]). Here we have a stronger requirement: every possible solution must be generated with *the same probability*[2].
- **Efficient:** Testing time includes generating time, since test cases are generated dynamically. Hence, to be of practical use, generators must maintain reasonable performance, in particular in large codebases that run tests frequently.

Property-based testing frameworks *à la* QuickCheck generally resort to rejection sampling to produce random values that meet specific constraints. While this technique meets the first two of the above requirements, it can lead to a significant overhead, in particular when a large portion of generated values are rejected. The impact on performances depends on the efficiency of the rejection process and the likelihood of rejection. We briefly recall how it works.

[1] We do not specify the actual representation of random states, as it is irrelevant to our discussion and depends on the underlying implementation of the generator.

[2] Some framework focus on building corner-case generators, generally defined in an *ad-hoc* fashion, in which case uniformity is irrelevant. This is a complementary approach that we are not focused on in this work.

2.1 Rejection Sampling

In the most general sense, rejection sampling is the approach that consists in generating samples in a super-set of the set of objects we are interested in, repeatedly until a sample lying in the set of interest is found. For instance, in order to generate a point inside the disk of radius 1 in $\mathbb{R}^2$, one could generate points (X, Y) in the square $[-1, 1]^2$ until $X^2 + Y^2 \leq 1$. In a constraint solving context, this means generating samples within the bounds of the problem until a sample that satisfy the constraints is found. Algorithm 1 illustrates this method.

Algorithm 1 Rejection sampling procedure.

```
1: function SAMPLE(V,D,C)
2:     candidate ← spawn (V,D)          ▷ draw at random in (V, D)
3:     if sat(candidate, C) then
4:         return candidate
5:     else
6:         return sample(V,D,C)
```

This algorithm takes as input a set of variables $\mathcal{V}$, their associated range of values $\mathcal{D}$, and a set of constraints $\mathcal{C}$. The `spawn` function generates a candidate at random (for some probability distribution $\mathbb{P}$) from the given variables and their domains. Then, the `sat` function evaluates whether the generated candidate satisfies all the constraints in $\mathcal{C}$. The procedure is called recursively until a solution is found.

Note that the number of iteration of this algorithm is a random variable. The probability of accepting a sample X at line 3 is the probability $P(X \in A)$ that X lies inside the solution space A. Provided that this probability is non-zero, this algorithms terminates, and its number of iterations follows a geometric law of parameter $\mathbb{P}(A)$. It thus makes $\frac{1}{\mathbb{P}(X \in A)}$ iterations in expectation.

Finally, a key observation is that Algorithm 1 implements the probability distribution $\mathbb{P}$ *conditioned* to only draw elements of the solution space A. In particular, if the `spawn` function draws uniform samples in the domain $\mathcal{D}$, then Algorithm 1 is a uniform sampler of solution. The contribution of the present paper is to devise a good `spawn` function that can guarantee that this algorithm is uniform while providing good performance.

3 Constraint-Based Generators

In this article, we use a constraint-based approach for random sampling. To achieve this, we rely on the general abstract solving method described in [21], which we summarize here.

Algorithm 2 constructs a cover of the solution space using abstract elements (e.g., boxes, octagons, polyhedra, etc.). This cover consists of two sets: inner elements (I) and outer elements (O). The set I under-approximates the solution set, while $I \cup O$ over-approximates it. The algorithm starts by initializing an abstract element and inserting it into O. It then iterates through the following steps: an element from O is selected, filtered, and, if it satisfies the constraints, added to I. If it does not, it is split into sub-elements that are reinserted into O.

As presented, this algorithm may not terminate. In practice, various termination criteria can be employed, such as limiting the size of the elements considered or the depth of the solving tree. Note that, in general, split elements can overlap without preventing the algorithm from terminating and producing correct results. However, in our case, ensuring they are non-overlapping is necessary for uniformity, as we will discuss later.

Algorithm 2 Abstract solving method.

```
1: function SOLVE(𝒳, 𝒟, 𝒞)          ▷ 𝒳:variables, 𝒟: domains, 𝒞: constraints
2:     I ← ∅                          ▷ inner solutions
3:     O ← {init(𝒟)}                  ▷ outer solutions
4:     while O ≠ ∅ do
5:         e ← select(O)
6:         e′ ← filter(e, 𝒞)
7:         if e′ ≠ ⊥ then
8:             if solution(e′, 𝒞) then
9:                 I ← I ∪ {e′}
10:            else
11:                O ← O∪ split(e)
12: return I, O
```

Although the algorithm is parametric with respect to the representation being used, we only use boxes in our implementation, as they enable straightforward and efficient uniform random sampling. Recall that given variables $v_1, \ldots, v_n$ over finite continuous domains $d_1, \ldots, d_n$, a box is defined as a Cartesian product of intervals within $d_1 \times \cdots \times d_n$. A random sample of a box is thus the Cartesian product of random samples of such intervals.

3.1 Constrained Based Sampling

In [29], authors build upon this constraint-solving method an algorithm for uniform sampling under constraints, as shown in Algorithm 3. It repeatedly selects an element e from either the inner or outer sets using a `select` function. Here, `select` performs a weighted choice based on the volume of the elements, with the largest elements having the best chance of being chosen. We discuss in the next section the design of a data structure that enables an efficient implementation of this function. The algorithm generates a candidate value i within e. If e belongs to the inner set or i satisfies the constraints $\mathcal{C}$, the algorithm returns i; it repeats these steps otherwise.

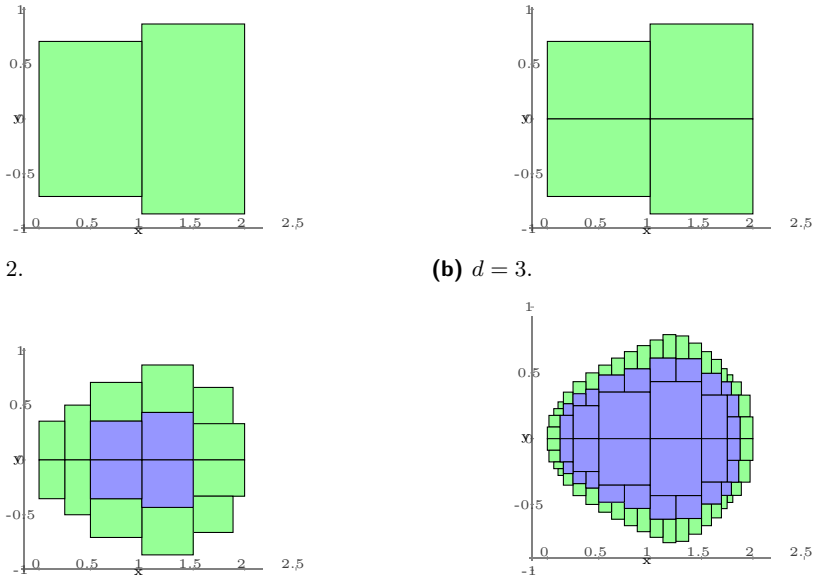
Algorithm 3 Random Sampling for Covers.

```
1: function GENERATE(inner,outer,𝒞)
2:     while true do
3:         e ← select(inner,outer)
4:         i ← spawn(e)
5:         if e ∈ inner ∨ sat(i,𝒞) then return i
```

This algorithm ensures uniform sampling under three conditions: the elements in $I \cup O$ must not overlap, points within each element e must be sampled uniformly $P(i \mid e) = \frac{1}{v(e)}$, and the probability of selecting an element e must be proportional to its volume ($P(e) = \frac{v(e)}{\sum_{e' \in I \cup O} v(e')}$). These conditions ensure that the sampling process remains uniform across the domain, giving all points in the union of elements an equal likelihood of being chosen.

▷ Claim 3. Algorithm 3 samples solutions uniformly. While uniformity was an implicit goal in [29], we provide a formal proof to establish it rigorously.

Proof. In appendix A. ◁



(a) $d = 2$. **(b)** $d = 3$. **(c)** $d = 5$. **(d)** $d = 8$.

Figure 1 Solving state obtained for depth d of resolution for a problem with two variables constrained by $x^2 + 4y^2 - 4 \leq 0$ and $2y^2 - x \leq 0$. Darker boxes indicate inner elements.

Refining the solving process can potentially reduce the rejection rate – though not always – but it never increases it. However, this refinement inevitably increases the selection time. At some point, this may even become counterproductive; when the increase in selection time exceeds the gains obtained from reducing the rejection rate, further refinement is no longer beneficial. Our goal is to find the best trade-off between constraint solving and rejection sampling, minimizing unnecessary exploration while ensuring efficiency.

3.2 Efficient selection of an abstract element

In order to implement the `select` function, we need a data structure for storing a collection of abstract elements that supports:

- efficient sampling of an elements with probability proportional to its volume;
- and (we will see later), an efficient way to replace an abstract element with a collection of smaller elements.

In [29], sorted list in decreasing order of volume are used for element selection, so that the elements with highest probability are met faster during sampling. However, if elements are of the same size, this approach offers no advantage, as all selections become equally likely. Also, they do not require to update their structure in their work. We can do better by arranging these elements in a binary tree. The idea is to store the abstract elements at the leaves of the tree and to maintain, in every node, the sum of the volumes of all the leaves below that node. Using such a data structure, drawing a abstract element proportional to its volume corresponds to drawing a uniform real variable between 0 and the total volume of the tree, and to recursively descend in the tree, choosing between the left and right child based on their weights. This is illustrated in Algorithm 4.

Here, the cost of selecting an element is proportional to the length of the path from the root to the selected leaf. Given the volume of every abstract element (and thus their probability of being drawn), there is an optimal way to arrange these elements in the tree

Algorithm 4 Random generation of an abstract element using the tree data structure.

```
function randomTreeSelect(T)
    r ← Unif(0, volume(T))
    return treeSelect(r, T)
```

Require: $0 \leq r < \text{volume}(T)$

```
function treeSelect(r,T)
    if T is a leaf then return the abstract element stored in T
    (T_L, T_R) ← the children of T
    if r < volume(T_L) then return treeSelect(r, T_L)
    else return treeSelect(r − volume(T_L), T_R)
```

in order to minimise the expected cost of the generation. Information theoretic results tell us that the expected path length between the root and the sample is lower bounded by the entropy of the probability distribution, that is

$$\mathcal{H} = \sum_x p_x \ln \frac{1}{p_x}$$

where the sum ranges over the abstract elements stored in the tree, and p_x is the sampling probability of element x. The optimal way to organise the tree in order to remain close to this lower bound is to use a *Huffman tree* [16]:

- start from a collection of leaves,
- iteratively pair the two smallest elements of the collection as a binary node,
- this process terminates when every leaves belong to the same tree.

A tree built this way has the property that its expected path length to a leaf is at most $\mathcal{H}+1$.

Unfortunately, the good properties of Huffman trees are hard to maintain efficiently when we update the collection of abstract elements, which we need for in the adaptive algorithm presented in Section 3.3. To circumvent this issue, we use the following heuristic:

1. before doing any sampling, we do a first solving pass until a certain depth;
2. after this pre-processing, we construct a Huffman tree T based on the volumes of the resulting elements;
3. then, during the iterative sampling process, every time we need to split an abstract element e into a collection of smaller elements $(e_1, e_2, \ldots, e_p)$, we construct a Huffman tree T' for $(e_1, e_2, \ldots, e_p)$ and we replace the leaf e in T with T'.

It is worth noting that the replacement of the last sampled leaf can be optimised by keeping a pointer to this last leaf, rather than traversing the tree a second time to find it.

The initial Huffman tree T constructed at step 2 thus potentially evolves away from its optimal shape as we update leaves. However, since the elements that have the highest probability to be selected (and thus split) are the bigger ones, we expected our heuristic to maintain some balance in the tree. Our benchmarks, presented in Section 6, seem to show that this data structure performs well in practice. An interesting algorithmic problem would be to investigate the real performance of this idea, and potentially find a better data structure, which is a work in progress.

3.3 Incremental and Adaptive Generators

From a practical point of view, we cannot really replace an inefficient sampler with a generator based on an extremely expensive constraint solving mechanism. This would simply replace a slow generation speed, by a faster one preceded by a huge overhead due to solving time. This is especially prejudicial when the generator is only used a handful of time.

Iterative solvers are commonly used when dealing with complex constraint satisfaction problems where finding an exact solution is computationally infeasible or impractical. Instead of attempting to solve the entire problem in one step, iterative solvers work incrementally, refining the solution repeatedly until a certain termination criterion is met.

We therefore propose to use constraint resolution mechanisms parsimoniously: first, we will target in priority on certain parts of the problem, identified on the fly, whose filtering can greatly improve the performance of random generation. Second, we try to amortize the cost of the resolution steps (filtering and exploration) during the generation. For example, every n time a generator is called, we can perform a resolution step. Thus, the more a generator is used, the more it improves. To achieve this, the generator embeds an internal solving state that evolves with each step. This state retains information from previous resolution steps, allowing the generator to improve progressively rather than starting from scratch each time.

Algorithm 2 revolves around two primary decisions: selecting which element to refine and determining how to refine it. On the one hand, intuitively, the focus should be on larger elements with higher rejection rates, as they are more critical to the rejection rate and, thus, the efficiency of the whole process. On the other hand, sampling also requires choosing an element, with larger elements being more likely to be selected. This leads to the question of whether sampling's focus on larger elements can be used to help guide the solving process.

For each element in our cover, we track the number of times it was selected and record the number of times it successfully produced a sample. We also track the total number of successes and failures for the whole cover. When our algorithm fails to produce a valid solution within a selected element, we must then decide whether it is necessary to split it or not. To do this, we base our choice on its acceptance rate. If it falls below the global acceptance rate, we proceed to refine that element, meaning we split it and replace it within the cover with the resulting sub-elements. Otherwise, it is left unchanged. This dynamic adjusting of the refinement process can lead to a faster convergence and avoids over-splitting, which can deteriorate the element selection procedure. This is illustrated by Algorithm 5.

Algorithm 5 Sampling-Guided solving.

```
1: function GENERATE(inner,outer,C)
2:     while true do
3:         e ← select(inner,outer)
4:         i ← spawn(e)
5:         if e ∈ inner ∨ sat(i,C) then
6:             return i
7:         else if rate(e) < global(inner,outer) then
8:             refine e
```

4 Graph Representation for Random Sampling

The constraint hypergraph of a constraint satisfaction problem is a hypergraph in which the vertices correspond to the variables, and the hyperedges correspond to the constraints. Hypergraph representations can be integrated with various constraint-solving algorithms, such as backtracking, constraint propagation, and local search. The hypergraph structure guides the search for a solution. For our needs, an interesting idea is the detection of connected components. Indeed, the partition of the constraint hypergraph into connected components correspond to a partition of the problem into statistically independent sub problems. This means that the sampling for each component can be decorrelated from the

sampling of the others. In other words, if the constraint graph consists of two (or more) connected components, it is then possible to generate solutions independently in the different components, and then combine the results. Intuitively, leveraging this independence should speed up the random generation process. We also give a heuristic argument.

The acceptance rate of a set of constraints $\mathcal{C}$ is the probability that every constraint in $\mathcal{C}$ accepts a random sample. We note denote by $P_{\mathcal{C}}$ this probability. The expected number of rejections before a sample satisfies all the constraints in $\mathcal{C}$ is thus $\frac{1}{P_{\mathcal{C}}}$. When there are two (or more) disjoint components in the graph, the set $\mathcal{C}$ can be split into two independent subsets, $\mathcal{C}_1$ and $\mathcal{C}_2$, with respective supports $\mathcal{V}_1$ and $\mathcal{V}_2$ and we have $P_{\mathcal{C}} = P_{\mathcal{C}_1} P_{\mathcal{C}_2}$. Leveraging the independence of the variables in $\mathcal{V}_1$ and $\mathcal{V}_2$ by performing the rejection sampling independently for the two components, yields a expected number of rejections of the order of

$$\frac{1}{P_{\mathcal{C}_1}} + \frac{1}{P_{\mathcal{C}_2}} \qquad \text{instead of} \qquad \frac{1}{P_{\mathcal{C}_1} P_{\mathcal{C}_2}}$$

which is largely smaller when the constraints $\mathcal{C}_i$ are hard to satisfy, *i.e.* when the $P_{\mathcal{C}_i}$ are small. Of course, a more precise analysis would require to take into account the cost of sampling a leaf in the Huffman tree and the cost of generating a uniform sample inside an abstract element. However, we can already foresee that splitting the Huffman tree into the two trees for the two connected components, and reducing the number of variables will only moderately affect the performance in the algorithm. We can conclude with confidence that leveraging the independence of the components, will be beneficial.

In practice, at initialization, the CSP is divided into (disjoint) connected components, and a sampling cache is associated with each one. This cache stores the partial samples generated for each component, allowing the algorithm to reuse previously computed samples and avoid redundant computations.

We benefit from this idea by using the cache to identify and prioritize the solver's efforts on components that exhibit high rejection rates. Empty caches indicate difficulty in finding valid samples, encouraging the solver to select a variable to split in these components. By adapting to the performance of individual components, the solver converges toward a locally optimal configuration.

4.1 The *bridge-first* split

The iterative process of splitting and filtering gradually ensures that certain constraints are locally satisfied, allowing them to be removed from the graph. Whenever a constraint is removed, the solver checks whether its removal disconnects the graph. When this is the case, the graph is split into connected components, enabling their independent handling.

Constraints become redundant when their validity is guaranteed by current domain assignments. Our implementation recomputes connected components whenever a constraint is removed by the solver so as to be able to exploit the independence of the components.

To achieve this, we develop a propagation scheme and an exploration heuristic that help reduce the connectivity of the constraint graph. For each edge, we maintain information on whether it is a bridge or not. A bridge is an edge whose removal increases the number of connected components in the graph. We identify bridges using Tarjan's algorithm [25]. Tracking which edges are bridges is useful for guiding exploration, since variables that are part of a bridge are particularly interesting for splitting. When a constraint is removed, if it was a bridge, the connected components (CCs) are recomputed. Otherwise, the removal is checked to determine whether it introduced new bridges, without recalculating the connected

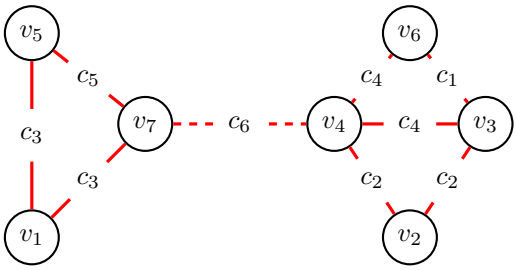


Figure 2 Deletion of the constraint c_6 leads to two independent CSPs.

components. If a bridge is detected, we attempt to eliminate it, as its removal would disconnect the graph and thus improve the sampler's performance. This is done by prioritizing the variables in the bridge's support during splitting steps.

5 Constraint Based Testing with GeGen

We have implemented the techniques presented in this paper in a prototype called *GeGen*, which stands for ***Ge****nerator* ***Gen****erator*. PBT frameworks like QCheck [4], which is the OCaml port of QuickCheck, provide generator for atomic types (bool, ints, floats, ...) and combinator for composite type (pairs, tuples, ...). In presence of recursive types, the traditional approach is to provide the user with a generator of *sized values*, that is user has to provide a size, and the generator builds values with that specific size. Most implementations lack robust support for testing scenarios that require constraints over the generated inputs. *GeGen* bridges this gap by introducing variable generators and constraints, enabling the easy integration of constraint-solving capabilities in a PBT framework. For instance, QCheck provides the function find_example whose signature is given below. Given a generator of a values of type t, it builds a generator of values of t that satisfy a certain predicate

```
find_example : ('a -> bool) -> 'a Gen.t -> 'a Gen.t
```

If a value satisfying the predicate `f` is found (within a certain number of tries), it is returned. *GeGen* mimics this approach by providing an API that users can manipulate similarly, however the inner mechanism differ notably as we build, solve and sample from a CSP during the generation process.

5.1 GeGen's Generators

GeGen's generators differ fundamentally from traditional QuickCheck-style generators as they produce *symbolic variables* rather than concrete values. Traditional property-based testing frameworks generate fixed values like random integers or floats. In contrast, GeGen's generators operate at a symbolic level which makes possible constraint composition and solving. For instance, instead of directly generating a number, *GeGen* creates a symbolic variable that represents the number and associates it with a range of possible values. This symbolic variable acts as a declaration on the solver's side. Additionally, instead of applying a predicate to verify wether a value satisfies or not the property being tested, GeGen's predicates impose constraints on this variable such as "*the number must be even*" or "*the number must be less than another variable*" that are collected to construct a CSP. Finally, we apply the previously discussed solving and sampling techniques to the constructed CSP. For example, to generate circles within a given square in QCheck, we can use the following code:

```
find_example
  (pair (pair int int) int)
  ~f:(fun ((x, y), r) ->
    r > 0 && x >= r && x <= 100 - r && y >= r && y <= 100 - r && r >= 10)
```

The find_example function generates random integer values for x, y, and r, then applies the given predicate to check whether the generated circle satisfies the constraints. This process repeats until a valid example is found or the search limit is exhausted. Note that here, int is an atomic generator that produces random integers and pair is a combinator that constructs a generator of pairs from two base generators.

The interest of our approach is that the same code, when linked against our library, will yield identical results but in a significantly more efficient manner, and using a fundamentally different mechanism. Instead of directly sampling random values for x, y, and r, we constructs a symbolic representation of the problem. From the solver's perspective, a constraint satisfaction problem (CSP) with three floating-point variables v_1, v_2 and v_3 is formulated. However, this approach abstracts certain details about the algebraic structure of the type. Therefore, a reconstruction function is designed alongside the CSP, so that once a sample s is drawn, it can be re-assembled to a value of the correct type. In this particular case, the function is $f(s) = ((s(v_1), s(v_2)), s(v_3))$. It ensures that once the solver produces a valid assignment for the symbolic variables, the corresponding concrete values are reconstructed in a way that respects the intended algebraic structure. Finally the predicate, instead of returning a truth value will actually build the equivalent constraint system. In essence, this allows us to decouple the problem-solving phase from the data generation phase (at the API level) while ensuring that the generated values maintain the correct type.

5.2 GeGen's language

The constraint language used in *GeGen* supports arithmetic and boolean expressions. It includes: arithmetic operation (addition, multiplication, etc.), Boolean logic (comparisons, conjunctions, disjunctions, negations), and variables. We build upon those generators for more complex types such as tuples and lists by composing the different elements of a generator.

GeGen extends standard arithmetic and Boolean operators by overriding them to facilitate the construction of constraints, making it possible to define a CSP in a manner that closely resembles programming its logical predicate counterpart. This intuitive approach aligns the construction of constraints with familiar programming paradigms, simplifying the transition from predicates to constraints.

For instance, consider a predicate that verifies whether a list is sorted. This predicate can naturally be expressed in a functional style as follows:

```
let rec is_sorted = function
| [] | [x] -> true
| x1 :: x2 :: rest -> x1 <= x2 && is_sorted (x2 :: rest)
```

This predicate can be applied to concrete lists of integers to determine whether they are sorted. This is what is done when doing rejection sampling. Using GeGen, the predicate takes another meaning as it operates on lists of variables to dynamically construct a CSP. The overloaded operators automatically translate the predicate logic into corresponding constraints. For example:

```
open GeGen
let rec is_sorted = function
| [] | [x] -> true_
| x1 :: x2 :: rest -> x1 <= x2 && is_sorted (x2 :: rest)
```

Here, the open **GeGen** directive brings in scope the operators <= and && (among others), so that instead of computing a boolean value, the is_sorted function now builds a constraint. Beside that the only difference is the use of the symbolic constraint true_ instead of the builtin OCaml boolean true, which can't be avoided as OCaml does not permit the overriding of litterals. This approach preserves the logical structure and readability of the predicate while generating a constraint representation that can be used to solve CSPs.

6 Experiments

Our implementation is written in OCaml in a functional style. We use the default *pseudo-random number generator* (PRNG) from the OCaml standard library, an instance of the LXM [24] family of PRNGs. *GeGen* relies on the *AbSolute* solver [21] to handle constraints. This solver provides most of the necessary functionalities for uniform random sampling, including volume measurement and efficient space paving with large elements [30]. It guarantees a non-overlapping solution cover, simplifying uniform distribution construction. Our implementation is open-source and available at:
`https://osf.io/u4r5q/files/osfstorage?view_only=84af8ee65a6c495d98cb7b7bfad8c54b`.

6.1 Benchmark

We have written generators using our framework for several realistic examples: the `convex` problem defines the predicate for star-convex polygons, *i.e.* the vectors corresponding to two consecutive edges of the polygons have a negative cross-product. The `diagdom`, `idempotent`, `symmetric` and `orthogonal` problems, define the predicates for some well known classes of matrices. The `distrib`, `itvcover` and `sorted` represent respectively the predicate for valid distribution of probability *i.e.*, list of positive values that sum to one, interval-covering list that is a list that contains a set of intervals that cover a specific range without gaps and finally the set of sorted lists in increasing order. We compare our approach against QCheck's standard rejection sampler, as it serves as a baseline for property-based testing. This comparison is relevant since we have designed an API that allows users to write the same specifications while seamlessly generating samples using our method instead.

Table 1 presents the number of samples generated by *GeGen* and QCheck for various problem instances. Each row corresponds to a specific problem. The first column lists the problem names. The second column indicates the problem size parameters, denoted as $|\mathcal{V}|, |\mathcal{C}|$. The remaining columns report the number of generated samples for different time constraints (0.1, 1, and 2 seconds of generation time), with results shown separately for *GeGen* and QCheck. Best result for each are shown in bold font. The problem sizes were selected to ensure that trends in sample generation could be clearly observed while maintaining a fair evaluation of both methods. The measurements were conducted on a machine with a 12th Gen Intel(R) Core(TM) i5-1235U processor and 15 GiB of RAM.

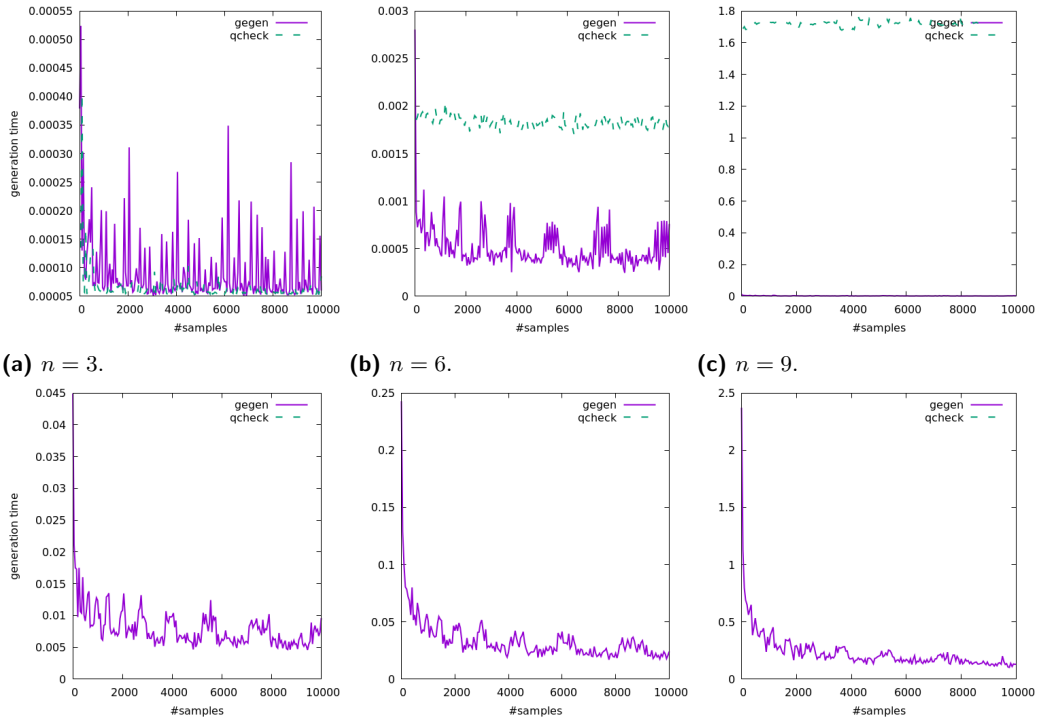
6.2 Results Analysis

Table 1 Number of samples per generation time

		GeGen			QCheck		
problem	$\lvert\mathcal{V}\rvert, \lvert\mathcal{C}\rvert$	0.1	1	2	0.1	1	2
sortedlist4	4,3	7076	292589	621378	**17999**	**527764**	**1107631**
sortedlist8	8,7	**410**	**32985**	**76783**	9	1047	2191
sortedlist12	12,11	**28**	**4891**	**12421**	0	0	0
convex3	6,3	**752**	**54829**	**114298**	5	415	868
convex4	8,4	**203**	**9182**	**19997**	0	0	2
convex5	10,5	**82**	**3526**	**7357**	0	0	0
diagdom3	9,4	823	**39282**	**83120**	**1052**	25354	52677
diagdom4	16,5	**141**	**6454**	**13680**	0	25	54
diagdom5	25,6	**20**	**984**	**2176**	0	0	0
itvcover3	6,5	**5522**	**225252**	**477677**	0	29	62
itvcover4	8,4	**4258**	**132492**	**279720**	0	1	3
itvcover5	10,9	**2724**	**87048**	**174027**	0	0	0
symmetric2	4,1	5617	**428439**	**901930**	**7842**	242921	493929
symmetric3	9,3	**7178**	**233311**	**490004**	0	1	3
symmetric4	16,6	**2176**	**136995**	**280146**	0	0	0
orthogonal2	4,5	**1457**	**59079**	**125073**	0	0	0
orthogonal3	9,8	**132**	**5110**	**10693**	0	0	0
orthogonal4	16,12	**3**	**596**	**1329**	0	0	0
idempotent2	4,10	**1494**	**93988**	**205829**	0	15	33
idempotent3	9,29	**35**	**3590**	**8295**	0	0	0
idempotent4	16,76	0	**41**	**158**	0	0	0

The results indicate that *GeGen* consistently outperforms QCheck, generating significantly more samples across nearly all problem types and sizes. While QCheck performs adequately for smaller problem instances, its performance drops as problem complexity increases, often failing to generate any samples for larger cases. In contrast, *GeGen* scales effectively, maintaining high sample generation rates even for more complex problems. Also, the performance of the rejection sampler remains stable over time while the ones of the generator built using *GeGen* improves as the process progresses. This trend is illustrated in Figure 3, which depicts the time in seconds (y-axis) required to generate the i-th sample (x-axis), for 10000 samples. *GeGen*'s times (solid line), and QCheck's (dashed line) are shown on a single problem, sorted lists of size n, to highlight this feature of our generators, although the same behaviour can be noticed on problems of table 1. Notable spikes in the curves correspond to garbage collection cycles, which are particularly apparent for small values of n.

Figure 3a demonstrates that for small list sizes, QCheck and *GeGen* exhibit similar performance. In fact, QCheck marginally outperforms *GeGen*, as the rate of sorted lists among lists of size 3 is relatively high. However, as n increases, the rejection rate also rises, causing *GeGen* to surpass QCheck. By the time n=9n=9, *GeGen* is already outperforming QCheck by three orders of magnitude, as shown in Figure 3c. Note that, due to the scale, the *GeGen* line appears very close to the x-axis. For larger values of n, rejection sampling via QCheck becomes impractical, and thus only the results for *GeGen* are presented. This figure shows that the incremental nature of our generators allows them to scale effectively, handling larger n more efficiently.



(a) $n = 3$. **(b)** $n = 6$. **(c)** $n = 9$.

(d) $n = 12$. **(e)** $n = 15$. **(f)** $n = 20$.

Figure 3 Evolution of the generation time (in seconds) for sorted list of size n, with both QCheck and *GeGen*.

Uniformity Validation. We have validated the uniformity of our generators by comparing their distributions with those generated by a rejection sampler. The variation between the distributions is measured using a chi-squared test, which quantifies how far apart the distributions are. Our results confirm that our generators produce distributions similar to the rejection sampler, indicating they are uniform.

7 Related Works

Random sampling of solutions of constrained systems is a well studied subject and the literature is full of techniques for SAT [28, 23, 10] models, and some results for CP models exists, but mainly for finite domains CSPs [7, 11, 13, 26, 15, 22].

Closer to our work are [2, 17, 29, 5]. In [2], the author present a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. In order to speed up random generation they use the lazy behaviour of the predicate to know its result on sets of values, rather than individual values. Once they have computed a set of values for which the predicate is going to return false, they remove all of these values from the original set. They implement this by relying on Haskell's call-by-need semantics which applies the predicate to a partially-defined value. This can be seen as an *ad-hoc* use of consistency on a partial assignment. Our work goes further by not only integrating a constraint resolution engine into the random generator, but in addition, this engine is dedicated to the manufacturing of relevant representations for random sampling.

In [29], the authors propose a constraint-based generation framework where constraint solving, which can be costly, is handled at compilation time via preprocessing, while uniform sampling occurs at runtime. While their approach is similar to ours, we find ours more practical as it is incremental and does not require preprocessing. Additionally, they do not focus on tuning the solving method for uniform sampling, instead fixing the solver's depth and size in advance, whereas we adapt constraint solving based on the sampler's output. More importantly, their approach cannot handle recursive types, a limitation our method overcomes effectively. In [5], a Constraint Logic Programming approach for Property based testing for Erlang programs, the authors generate random tests for complex properties involving Modified Condition/Decision Coverage, pattern matching, and higher-order functions. This is equally useful but orthogonal to our wors as their focus is put on code coverage and not of uniformity. A similar idea, in the field of computational statistics, is Adaptive Rejection Sampling [19, 18] (ARS). The idea is to use a piecewise linear density function instead of a single uniform envelope density function. Each time a sample has to be rejected, the rejected value can be used to improve the piecewise approximation of the targeted distribution. This therefore reduces the chance that the next attempt will be rejected. This can however only be applied for sampling from for specific families of densities.

Further from our field, other random sampling frameworks include the so-called recursive method from Nijenhuis and Wilf [20] and the framework of Boltzmann sampling [8]. These frameworks provide a generic and efficient way to sample structured data such as trees, algebraic data types, etc. but are not suitable for numerical data. Finally, Monte-Carlo Markov Chains (MCMC) are another well-known tool for sampling discrete structures [14]. However, here again, tuning a Markov Chain to make it efficient requires some specific knowledge on the objects to sample, which make it unsuitable for generically sampling into an arbitrary CSP solution space.

8 Conclusion

In this work, we have developped a method for building generators that are unifom, efficient, and incremental. Our approach can be summarized as the gradual transition from rejection sampling to a form of constraint-based one. Users have a generator that adapts to their needs: If the generator is rarely used, there is no need to spend time solving complex problems, and the underlying search space remains largely unprocessed by the solver. As a result, the generator behaves more like a rejection sampler. Conversely, if the generator is heavily used, the solver will more aggressively filter the search space, aiming to amortize the resolution cost through improved generation speed. The techniques we propose holds for both finite and continuous domains as the number of solutions in a given search-space, whether it is finite or not, is abstracted by the notion of weight. Also, our method for splitting the CSP into independant pieces, which greatly improves our samplers, can be reused in other context. For example, it makes the parallelization of the solving process straightforward since each component can be treated independently.

Our approach has several areas for improvement. We do not backtrack, assuming that our exploration choices always enhance the sampler, which is not guaranteed. Our generator synthesis is general and may be less efficient than constraint-specific methods. Constrained data structures often exhibit symmetries, suggesting that symmetry-breaking techniques could be a natural extension. Additionally, non-uniform methods may offer a better balance between bug-finding effectiveness and computational cost. While uniformity can be expensive, surjective generators may provide sufficient coverage. Finally, incorporating corner-case analysis into solution sampling could further improve results in future work.

References

1 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of SAT witnesses. *CoRR*, abs/1304.1584, 2013. arXiv:1304.1584.

2 Koen Claessen, Jonas Duregård, and Michał H. Pałka. Generating constrained random data with uniform distribution. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 18–34, Cham, 2014. Springer International Publishing.

3 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. doi:10.1145/357766.351266.

4 Simon Cruanes. Qcheck: Property-based testing for ocaml. https://github.com/c-cube/qcheck. Accessed: 2025-04-02.

5 Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. Property-based test case generators for free. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 186–206, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-31157-5_12.

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

7 Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *AAAI/IAAI*, 2002.

8 Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing*, 13(4-5):577–625, 2004. doi:10.1017/S0963548304006315.

9 Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Random generators for dependent types. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, pages 341–355, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

10 Stefano Ermon, Carla P. Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. *CoRR*, abs/1210.4861, 2012. arXiv:1210.4861.

11 Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In *International Conference on Principles and Practice of Constraint Programming*, pages 711–715, September 2006. doi:10.1007/11889205_56.

12 Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 274–289, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30201-8_22.

13 Arnaud Gotlieb and Matthieu Petit. A uniform random test data generator for path testing. *Journal of Systems and Software*, 83(12):2618–2626, 2010. TAIC PART 2009 - Testing: Academic & Industrial Conference - Practice And Research Techniques. doi:10.1016/j.jss.2010.08.021.

14 W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. URL: http://www.jstor.org/stable/2334940.

15 Kun He, Chunyang Wang, and Yitong Yin. Sampling lovász local lemma for general constraint satisfaction solutions in near-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 147–158, 2022. doi:10.1109/FOCS54457.2022.00021.

16 David Albert Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

17 Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158133.

18 Luca Martino. Parsimonious adaptive rejection sampling. *Electronics Letters*, 53, June 2017. doi:10.1049/el.2017.1711.

19 Luca Martino and Joaquín Míguez. A generalization of the adaptive rejection sampling algorithm. *Statistics and Computing*, 21:633–647, 2010. doi:10.1007/S11222-010-9197-9.

20 Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms: For Computers and Hard Calculators*. Academic Press, Inc., USA, 2nd edition, 1978. doi:10.1016/C2013-0-11243-3.

21 Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 434–454, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-35873-9_26.

22 Gilles Pesant, Claude-Guy Quimper, and Hélène Verhaeghe. Practically uniform solution sampling in constraint programming. In Pierre Schaus, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 335–344, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-08011-1_22.

23 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)*, pages 240–251, 2019. doi:10.1109/ICST.2019.00032.

24 Guy L. Steele Jr. and Sebastiano Vigna. Lxm: Better splittable pseudorandom number generators (and almost as fast). *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485525.

25 R.Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161, 1974. doi:10.1016/0020-0190(74)90003-9.

26 Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution Sampling with Random Table Constraints. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 56:1–56:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2021.56.

27 Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8):514–534, 2007. doi:10.1016/j.artint.2007.04.001.

28 J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using bdds. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, pages 584–589, 1999. doi:10.1109/ICCAD.1999.810715.

29 Ghiles Ziat, Matthieu Dien, and Vincent Botbol. Automated Random Testing of Numerical Constrained Types. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 59:1–59:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2021.59.

30 Ghiles Ziat, Marie Pelleau, Charlotte Truchet, and Antoine Miné. Finding solutions by finding inconsistencies. In *CP 2018 - 24th International Conference on Principles and Practice of Constraint Programming*, pages 1–16, Lille, France, August 2018. URL: https://hal.science/hal-01885769.

A Proof of correctness of Algorithm 3

Proof for Claim 3

Proof. Assume `spawn(e)` spawns points uniformly within an element e, i.e., for any $i \in e$, $P(i \mid e) = \frac{1}{v(e)}$, where $P(i \mid e)$ is the probability density function describing the likelihood of sampling a point i within the element e, and $v(e)$ is the volume of e. Let $\mathcal{S}$ be the set of all solutions satisfying constraints $\mathcal{C}$. The probability of sampling $i \in (\mathcal{I} \cup \mathcal{O})$ involves selecting an element $e \ni i$ and spawning i uniformly within e. The selection probability of e

is proportional to its volume:

$$P(e) = \frac{v(e)}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')}.$$

Thus, the joint probability of selecting e and spawning i is:

$$P(i \text{ is spawned}) = P(e) \cdot P(i \mid e) = \frac{v(e)}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')} \cdot \frac{1}{v(e)} = \frac{1}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')}.$$

Now, a point sampled with this process, without proper rejection, may not belong to $\mathcal{S}$. If $e \in \mathcal{I}$, all points $i \in e$ satisfy $\mathcal{C}$, so i is always accepted. If $e \in \mathcal{O}$, i is accepted only if $i \in \mathcal{S}$, and otherwise we repeat the sampling process. It follows that Algorithm 3 returns a particular solution $i \in \mathcal{S}$ either if:

- it samples i at the first attempt,
- or if the first attempt fails to produce a solution and i is sampled in a subsequent attempt.

The probability of sampling a particular solution $i \in \mathcal{S}$ is thus

$$\begin{aligned} P(i) &= \frac{1}{\sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')} + P(\text{fail})P(i) \\ &= \frac{1}{(1 - P(\text{fail})) \sum_{e' \in \mathcal{I} \cup \mathcal{O}} v(e')} \end{aligned}$$

where $P(\text{fail})$ is the probability that a uniform element in $\mathcal{I} \cup \mathcal{O}$ does not satisfy $\mathcal{C}$. This is constant for all $i \in \mathcal{S}$, proving uniformity.

We wrote this proof in a discrete setting but a similar proof can be written in the continuous. ◁