Microsoft Research

Diversely Enumerating System-Level Architectures

MSR Technical Report MSR-TR-2013-56

Diversely Enumerating System-Level Architectures

Ethan K. Jackson
Microsoft Research, USA
ejackson@microsoft.com

Gabor Simko
Vanderbilt University, USA
gabor.simko@
isis.vanderbilt.edu

Janos Sztipanovits
Vanderbilt University, USA
janos.sztipanovits@
vanderbilt.edu

ABSTRACT

Embedded systems are highly constrained. System-level constraints, such as task partitioning problems and communication scheduling problems, are common, combinatorial, and fundamentally intractable. Though modern constraint solvers can help to synthesize constrained architectures, the architect's troubles do not end here: There may be (infinitely) many architectures satisfying system-level constraints. Multiple candidates must be examined and this is often infeasible for large solution spaces.

In this paper we describe an improved enumeration scheme, which still reaps the benefits of modern constraint solvers. The idea is to build a *diverse enumerator* around an unmodified constraint solver. A diverse enumerator uniformly draws equivalence classes of solutions. Such an enumerator is powerful because it allows unbiased enumeration of the space and can be used to make inferences about the space as a whole. This paper presents the theory, practice, and algorithms for diverse enumeration of architectures with system-level constraints.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Architectures, constraints, diverse enumeration, symmetry directed randomized partitioning, synthesis

1. INTRODUCTION

Embedded systems are highly constrained. System-level constraints, such as task partitioning problems [20, 11] and communication scheduling problems [12, 9, 8], are common, combinatorial, and fundamentally intractable. Fortunately, system-level constraints are often solvable using modern constraint solvers, such as SAT [21] or SMT [16] solvers, to compute reasonable architectures. Yet, the architect's troubles do not end here [22]: There may be (infinitely) many architectures satisfying system-level constraints, so the architect must examine many candidates to decide upon the most reasonable design. For large solutions spaces it may be infeasible to (automatically) examine and rank all solutions. Also, it may be computationally difficult to determine if one solution is equivalent to another.

In this paper we describe an improved enumeration scheme, which still reaps the benefits of modern constraint solvers. The idea is to build a *diverse enumerator* around an unmodified constraint solver. We formally define a diverse enumerator to be a stochastic process uniformly drawing equivalence classes of solutions. This is a stringent definition and dramatically different from other definitions of diversity [22, 16, 18, 4]. A diverse enumerator is powerful, because it allows the architect to treat enumeration as a random sampling of non-equivalent architectures. A diverse enumerator can be used to infer statistics about a whole solution space, or for examining a fraction of the space without fearing selection bias.

In this paper we develop a general theory and practice of diverse enumeration applicable to a wide range of engineering problems. This approach has been implemented in the FORMULA framework [13], built on top of the *Z3* SMT solver [5]. Our contributions are:

- **General Theory.** We develop a general theory for modeling enumerators and quantifying their diversity. We formalize biased enumerators using a novel application of *Wallenius Urns*. The *ideal diverse enumerator* is a special-case of a Wallenius Urn where equivalence classes are drawn with equal probability. Divergence from the ideal can be precisely quantified.
- **Practical Problem Class.** We identify a broad class of enumeration problems called *constrained graphs*. Constrained graphs are a unifying representation for expressing mapping, allocation, and configuration problems. We fix the equivalence relation on graphs to be graph isomorphism, which provides a strong notion of equivalence.
- **Effective Algorithm.** We present an algorithm, called *Symmetry Directed Randomized Partitioning* (SDRP) for diverse enumeration of non-isomorphic constrained graphs. SDRP uses state-of-the-art techniques to efficiently drive solvers into diverse regions of the search space. It handles the notoriously tricky task of encoding isomorphism classes.

- **Experimental Results**. We quantitatively compare SDRP with other approaches to diverse enumeration, and show its superiority. We experimentally show that unmodified solvers are highly-biased enumerators. Also, increasing randomization in low-level solver algorithms does little to correct this problem.

To our knowledge this is the first attempt to systematically define, quantify, and compare a probabilistically strong notion of diversity. For the architecture enumeration problems we consider here, there can be exponentially many equivalent architectures that would be judged as maximally diverse according to other approaches.

This paper is structured as follows: Section 2 presents enumeration problems and our general theory of enumerators. Section 3 introduces constrained graph problems using a software/hardware co-design example. Section 4 describes the SDRP algorithm for diverse enumeration of constrained graphs. Section 5 tests SDRP against several other enumeration algorithms. Finally, we conclude with related work and discussion in Sections 6 and 7.

2. A THEORY OF ENUMERATORS

2.1 Enumeration Problems

We begin with some basic definitions for discussing enumeration problems: Let $\mathcal{L}$ be a logic and $\varphi[\vec{x}]$ a well-formed formula of $\mathcal{L}$ with free variables $\vec{x} \stackrel{def}{=} (x_1, \ldots, x_n)$. A *valuation* θ is a function from variables to values. A valuation *satisfies* φ, written $\theta \models \varphi[\vec{x}]$, if replacing every x_i by $\theta(x_i)$ in φ results in a valid formula of $\mathcal{L}$. A *constraint solver* for $\mathcal{L}$ finds satisfying valuations to well-formed formulas.

Definition 1 (Enumeration Problem)**.** An *enumeration problem* is a quadruple $\mathcal{I} \stackrel{def}{=} (\mathcal{C}, \varphi[\vec{x}], decide, \sim)$ where:

1. $\mathcal{C}$ is a set of concrete outcomes.
2. $\varphi[\vec{x}]$ is a constraint over *variables* $\vec{x}$.
3. $decide : \Theta \to \mathcal{C}$ maps valuations of $\vec{x}$ to outcomes.
4. $\sim$ is an equivalence relation on outcomes.

A *solution* is an outcome obtained from a valuation that happens to satisfy the constraints. The solution space is all the solutions: $space(\mathcal{I}) \stackrel{def}{=} \{decide(\theta) \mid \theta \models \varphi[\vec{x}]\}$.

For convenience, we have factored enumeration problems into a constraint formula φ and a decider that chooses outcomes by how the constraints were satisfied. For example, a task partitioning problem is an enumeration problem where:

- $\mathcal{C}$ is all possible functions from tasks onto processors.
- $\varphi[\vec{x}]$ are resource constraints among tasks and processors.
- *decide* maps a valuation of variables to a placement function.

A candidate architecture is one the satisfying resource constraints. These solutions are usually found by solving φ with a constraint solver and then reconstructing the mapping from tasks onto processors (which is the job of *decide*). If CPUs are homogeneous, then many placements are behaviorally equivalent. Table 1 describes several architecture enumeration problems and highlights common equivalence relations.

2.2 Enumeration Processes

An *enumeration algorithm* takes as input an enumeration problem and then outputs a sequence of non-equivalent solutions. Our first goal is to construct stochastic models of non-ideal enumerators such that diversity can be defined, measured, and quantitatively compared. This approach is desirable, because constraint solvers are complex pieces of software employing heuristics, guesses and random restarts [2]. It is impractical to compare enumerators by examining their implementations. These definitions take into account the behavior of practical enumerators, but are solver-independent.

We model an enumeration algorithm as a function from enumeration problems to stochastic processes. Each stochastic process summarizes the search behavior of the algorithm on a given problem. We apply *probabilistic urns* to model complex search behaviors, and then use these urns to measure proximity to the ideal. To perform quantitative comparisons of real enumerators we shall fit urn models to observed search behaviors (as detailed in Section 2.4).

Probabilistic urns are rich, well-studied stochastic processes for modeling many phenomena [14]. An urn contains a finite set of elements X, which have attributes such as *color* or *weight*. Elements are sequentially drawn from the urn according to probabilities derived from the elements' attributes. If each element is put back in the urn after it is drawn (called *with replacement*), then this is a sequence of *independent events*. Otherwise, if elements are drawn *without replacement*, then this is a sequence of *dependent events*.

We use urns to model the search behavior of an enumerator on a problem $\mathcal{I}$ as follows:

- The elements in the urn are the equivalence classes $X \stackrel{def}{=} \{[s]_\sim \mid s \in space(\mathcal{I})\}$. Enumerating m solutions is modeled by drawing m elements, in sequence, from the urn.
- The enumerator may be biased; it may prefer to enumerate some elements over others. Bias is captured by a weight function ω assigning a positive real weight to every element. If $\omega(e) > \omega(e')$ then e has a higher probability of being drawn than e'.
- Constraint solvers use backtracking to incrementally search for the next solution. This means the same solution will not be returned again and the next solution is dependent on the current state of the search. Therefore, we assume elements are drawn *without replacement*; enumeration is a sequence of dependent events.

The qualities of our urns are a special case of a model due to *Wallenius*, which gives rise to *Wallenius' Multivariate Noncentral Hypergeometric distribution* [14]. The conditional probability distribution of Definition 2 is a specialization of the Wallenius distribution.

Definition 2 (Problem Urn)**.** A *problem urn* $\mathcal{U}(\mathcal{I}, \omega)$ is an urn with elements $X \stackrel{def}{=} \{[s]_\sim \mid s \in space(\mathcal{I})\}$, weight function $\omega : X \to \mathbb{R}_+$, and the following conditional probability distribution: Let $\mathbf{e}$ be the event of drawing an element from $\mathcal{U}(\mathcal{I}, \omega)$ when it contains only the elements $Y \subseteq X$:

$$Pr(\mathbf{e} = e \mid e \notin Y) \stackrel{def}{=} 0.$$
$$Pr(\mathbf{e} = e \mid e \in Y) \stackrel{def}{=} \frac{\omega(e)}{\sum_{e' \in Y} \omega(e')}.$$

Problem	$\mathcal{C}$	$\varphi[\vec{x}]$	*decide*	$\sim$
Task Partitioning [20, 11]	Deployments of tasks onto processors.	Resource and schedulability constraints.	Produces deployments from placements encoded as 0-1 variables.	Deployments differ only by processor identities.
Feature selection [17, 19]	Trees of features.	Feature interaction constraints.	Maps feature selection variables to a sub-tree of features.	Feature trees differ only by ordering.
Distributed controller synthesis [24, 15]	Automata connected by channels.	An (CTL*) invariant the controller must satisfy.	Reconstructs automata / topology from encoding of synthesis problem.	Synthesized automata are isomorphic.

Table 1: Several kinds of architecture enumeration problems with system-level constraints.

We simply write $\mathcal{U}$ when the context is clear. (For the remainder of this paper we assume all enumeration problems have a finite number of equivalence classes.)

The power of the Wallenius model is its ability to capture bias and competition between elements. The probability of drawing an element depends on its weight, but also on the weights of all the elements remaining in the urn. For example, if search is heavily biased towards a few solutions, then we expect to draw these early. After a few draws the bias vanishes because the remaining elements compete with similar weights. Thus, Wallenius urns can model various search dynamics through weight functions. To our knowledge this is the first time Wallenius urns have been used to model heuristic search.

The *unbiased urn* is a special case where every element has the same weight. In this case elements will always be drawn with equal probability.

Lemma 1 (Unbiased Urn)**.** *An urn is unbiased if the probability of drawing any element in the urn is $\frac{1}{|Y|}$, where Y is the set of elements currently in the urn. An urn is unbiased iff it has a constant weight function.*

An *enumeration process* models an entire algorithm by associating every problem with a problem urn.

Definition 3 (Enumeration Process)**.** An enumeration process Π is a function from problems to urns such that:

$$\Pi(\mathcal{I}) \mapsto \mathcal{U}(\mathcal{I}, \omega), \text{ for some } \omega.$$

The *ideal diverse enumeration process* Π_{ideal} maps every problem to an unbiased urn.

For an ideal diverse enumeration process the probability of drawing an equivalence class is the same for all classes. This is a statistically strong property guaranteeing unbiased sampling of the space.

2.3 Defining Bias

It is unlikely that any reasonable algorithm behaves ideally. In this case, its urn models can be used to compare with the ideal enumeration process. There are many techniques for comparing stochastic processes with uniform distributions, e.g. entropy or χ^2 statistics. However, these values are difficult to compute for Wallenius urns, because the underlying PMFs have exponential support in the size of the urn. Instead, we compute a simpler and intuitive likelihood ratio. Let $P_{max}(\mathcal{U})$ be the probability of the most likely sequence of draws, when drawing all elements from $\mathcal{U}$. Our bias number reflects how many times more likely $P_{max}(\mathcal{U})$ is compared to the unbiased urn. Computing $P_{max}(\mathcal{U})$ is a simple polynomial-time operation as shown below:

Lemma 2 (Most Likely Sequence)**.** *Consider the event of drawing all $|X|$ elements from $\mathcal{U}$. Let $e_1, \ldots, e_n$ be a ordering of elements such that $i \leq j$ implies $\omega(e_i) \geq \omega(e_j)$. Then the most probable sequence of $|X|$ draws has the probability $P_{max}(\omega)$:*

$$P_{max}(\omega) \stackrel{def}{=} \prod_{1 \leq i \leq n} Pr(e_i \mid e_i \in X - \{e_1, \ldots, e_{i-1}\}).$$

Lemma 2 observes that the sequence of draws with highest probability is any sequence which draws elements in non-increasing order of their weights.

Corollary. *If $\mathcal{U}$ is unbiased then every sequence of draws has maximal probability equal to $P_{unb}(|X|)$, where*

$$P_{unb}(n) \stackrel{def}{=} \frac{1}{n!}.$$

The *likelihood ratio* $\frac{P_{max}(\omega)}{P_{unb}(|X|)}$ indicates how many times more likely it is to draw a maximal sequence from $\mathcal{U}(\mathcal{I}, \omega)$ compared to an unbiased urn with the same number of elements. Because this ratio can be large, we define the bias to be the *log-likelihood ratio* normalized to the interval $[0, 1]$.

Definition 4 (Urn Bias)**.** The bias of $\mathcal{U}$ is the normalized log-likelihood ratio:

$$\beta(\mathcal{U}) \stackrel{def}{=} 1 - \frac{ln(P_{max}(\omega))}{ln(P_{unb}(|X|))}.$$

The larger the value of β the more biased the urn. Figure 1 shows several weight functions and their biases. Each function defines an urn with 32 elements labeled $1, \ldots, 32$. Function 1.I describes an unbiased urn; it has the smallest possible bias of 0. Functions 1.II/1.III have elements with decreasing exponentially/linearly spaced weights. The decreasing exponential has the largest bias, because the maximal sequence of draws is exponentially preferred. The step function 1.IV splits the elements into classes of high and low preference. Note that bias is a normalized log-likelihood ratio, so small differences in β correspond to exponential differences in the likelihood ratios.

2.4 Comparing Enumeration Algorithms

Suppose a process Π accurately models the behavior of an enumeration algorithm $\mathcal{A}$. Then, the proximity to the ideal

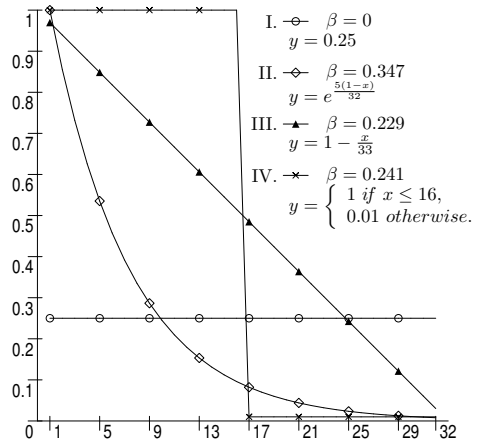

Figure 1: Several examples of weight functions and their bias. X-axis indicates element labeled $1, 2, \ldots, 32$; y-axis is $\omega(e)$.

is reflected in the average urn bias. (Recall for Π_{ideal} this average is zero.)

Definition 5 (Process Bias)**.** The process bias $\beta(\Pi)$ is the average of its urn biases.

The main challenge is accurately constructing Π by observing some runs of $\mathcal{A}$. In practice, we can only construct a partial specification of Π from a finite set of benchmark problems.

The overall bias of $\mathcal{A}$ is estimated by inferring urns from the search behavior of $\mathcal{A}$ on some problem instances. Unfortunately, the relationship between urn weights and probabilities is complex, but approximate inference of ω is possible. Suppose urns $\mathcal{U}$ and $\mathcal{U}'$ have weight functions $\omega = k \cdot \omega'$ for some $k > 0$, then the urns define the same stochastic process. In other words, urn weights can be rescaled without consequence. Let μ_i^m be the expected number of times e_i is drawn from $\mathcal{U}$ after m draws, then $\mathcal{U}$ can be reconstructed from the first-draw expectations $\mu_1^1, \ldots, \mu_n^1$.

Lemma 3 (First-Draw Correspondence)**.** *Suppose $\mathcal{U}(\mathcal{I}, \omega)$ has first-draw expectations $\mu_1^1, \ldots, \mu_n^1$. Let $\mathcal{U}'(\mathcal{I}, \omega')$ have weight function $\omega'(e_i) \stackrel{def}{=} \mu_i^1$. Then $\mathcal{U}'$ is the same stochastic process as $\mathcal{U}$.*

PROOF. For all e, the expected number of times e is drawn after one draw is just the probability $Pr(\mathbf{e} = e | e \in X)$. Therefore:

$$w'(e) = k \cdot w(e) \text{ for } k = \frac{1}{\sum_{e'} w(e')}.$$

□

If the draw expectations for the first draw are known, then the urn weights can be reproduced (up to an inconsequential scaling factor). In practice, m should be larger than 1 to observe the search behavior of an algorithm. However, since even our smallest benchmarks contain thousands of non-equivalent solutions, Lemma 3 can be generalized for large urns:

Proposition 4 (Small-Draw Proposition)**.** *Given $\mathcal{U}(\mathcal{I}, \omega)$, $m \ll |X|$, and expectations $\mu_1^m, \ldots, \mu_n^m$. Then $\mathcal{U}'(\mathcal{I}, \omega')$ approximates $\mathcal{U}(\mathcal{I}, \omega)$, where $\omega'(e_i) \stackrel{def}{=} \mu_i^m$.*

The Small-Draw Proposition states that if m is much smaller than $|X|$, then the expectations $\mu_1^m, \ldots, \mu_n^m$ serve as a reasonable approximation of ω. We shall apply the proposition without reservation, but check that inferred weights are good approximations of $\mathcal{A}$'s search behavior.

The Small-Draw Proposition reduces the problem of weight inference to estimating draw expectations. Algorithm 1 performs this task. The input is an enumeration algorithm $\mathcal{A}$, a problem $\mathcal{I}$, a draw size m, and a number of repetitions r. InferUrn($\mathcal{A}$, $\mathcal{I}$, m, r) repeatedly samples m non-equivalent solutions to $\mathcal{I}$ by Enumerate($\mathcal{A}$, m). $\mathcal{A}$ is "reset" between each repetition, meaning all state stored by $\mathcal{A}$ is reset to initial conditions and new random number seeds are randomly chosen. The observations are stored in 0-1 matrix S where rows are labeled by repetition number and columns by equivalence classes. Initially S contains 0 at every position. The output is a vector $\hat{\omega}$ estimating each μ_i^m by averaging the number of times e_i was observed per repetition.

Algorithm 1 InferUrn($\mathcal{A}$, $\mathcal{I}$, m, r)

Require: $0 < m \ll |X|$ **and** $r > 0$,
1: $S \leftarrow$ ZeroMatrix($[1, \ldots, r], X$)
2: **for** $1 \leq i \leq r$ **do**
3: Reset($\mathcal{A}$)
4: $draw \leftarrow$ Enumerate($\mathcal{A}, m$)
5: **for** $s \in draw$ **do**
6: $S[i,$ GetClass$(s, X)] \leftarrow 1$
7: **end for**
8: **end for**
9: $\hat{\omega} \leftarrow \frac{1}{r} \cdot$SumColumns($S$)
10: **return** $\hat{\omega}$

Due to finite sampling many elements may not be observed within r repetitions. The estimated weight for an unobserved element is zero, so the bias for $\hat{\omega}$ must be corrected to account for zero-weight elements. We employ the following correction, which penalizes for zero-weight elements:

Definition 6 (Estimated Bias)**.**

$$\hat{\beta}(\hat{\omega}) \stackrel{def}{=} 1 - \frac{ln(P_{max}(\hat{\omega}_+)}{ln(P_{unb}(min(r \cdot m, |X|)))}.$$

The numerator is computed for $\hat{\omega}$ restricted to elements with non-zero weight estimates ($\hat{\omega}_+$). It penalizes for unobserved elements, because maximal sequences have higher probabilities when fewer weights are considered. The denominator rescales the penalty to the maximum number of elements that could have been observed: $r \cdot m$ unless $r \cdot m > |X|$. In the interest of space, we shall not discuss the statistical properties of this estimator or the mechanisms to check the Small-Draw Proposition.

3. CONSTRAINED GRAPHS PROBLEMS

We now present a practical class of enumeration problems called *constrained graphs*. Constrained graphs are simple to define, yet general enough to unify many enumeration problems. They are implemented in the FORMULA framework [13]. Let $\mathcal{G}$ be the set of all finite graphs with labels as vectors of values.

Definition 7 (Constrained Graph)**.** A constrained graph $\mathcal{I}_H$ is a enumeration problem where:

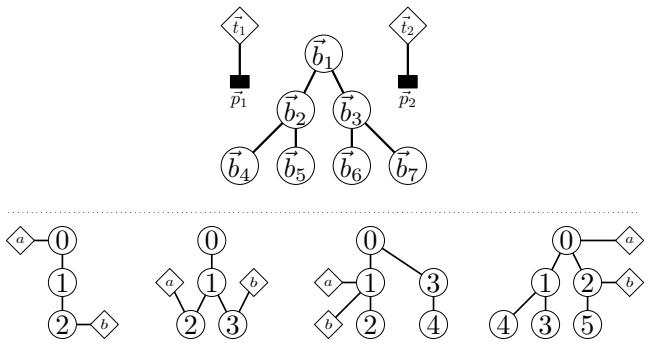

Figure 2: Top: Constrained graph for SW/HW co-design. Bottom: Examples of non-equivalent architectures.

- $H \stackrel{def}{=} (V_H, E_H)$ is a finite graph labeled by vectors of variables.
- Outcomes are graphs in $\mathcal{G}$ and $\sim$ is graph isomorphism.
- $decide(\theta)$ yields a graph $(\theta(V_H), \theta(E_H))$,

where $\theta(V_H)$ replaces each vertex label $\vec{y}$ with $\theta(\vec{y})$; $\theta(E_H)$ is similarly defined.

The graph H captures the structure of the enumeration problem. A concrete outcome is constructed by substituting the variables appearing in vertex labels with constants according to a valuation θ. This replacement may cause several nodes to receive the same final label, thereby folding H. A solution is an image of H formed by a satisfying valuation. We now illustrate constrained graphs for a software/hardware co-design problem.

3.1 Enumerating Bus Topologies

Consider the problem of designing a bus topology that can support the communication requirements of a set of tasks. In this simplified version of the problem, we desire a task partitioning and bus topology such that no bus is overloaded by the communication of requirements of the tasks place on it. Though this is clearly an NP-complete problem, it simplifies away many of the details of a complete communication scheduling problem. See [12, 8] for detailed investigations of the problem. We now give the complete encoding of this problem as a constrained graph.

Example 1 (Enumerating Bus Topologies)**.** The graph H has vertices representing tasks, buses, and placements, as shown in the top of Figure 2. The vertices labeled by variables $\vec{b}_i$ stand for names of buses, which are connected in a tree topology. Deciding that two buses have the same name, e.g. $\theta(\vec{b}_2) = \theta(\vec{b}_3)$, changes the structure of the final topology. The variables $\vec{p}_i$ decide where to place tasks t_i. For instance, if $\theta(\vec{p}_1) = \theta(\vec{b}_1)$ then task t_1 is placed at the root bus.

The constraint formula φ limits the legal bus topologies and the placements of tasks onto buses. First, we shall constrain buses to always form a tree topology of depth 3. This imposes the following constraints on bus labels: Buses at different depths in H (Figure 2, top) cannot have the same name:

$$\varphi_1 \stackrel{def}{=} \bigwedge_{depth(\vec{b}_i,H)\neq depth(\vec{b}_j,H)} \vec{b}_i \not\approx \vec{b}_j.$$

The function $depth(\vec{b}, H)$ is the depth of vertex $\vec{b}$ in H. For example, $depth(\vec{b}_1, H) \mapsto 0$. Buses with same names must have the same parents, otherwise the topology is not a tree:

$$\varphi_2 \stackrel{def}{=} \bigwedge \vec{b}_i \approx \vec{b}_j \Rightarrow parent(\vec{b}_i, H) \approx parent(\vec{b}_j, H).$$

The function $parent(\vec{b}, H)$ is the parent vertex in H. For example, $parent(\vec{b}_5, H) \mapsto \vec{b}_2$. Every task is distinct and must be assigned to a bus, which means every placement receives a valid bus label:

$$\varphi_3 \stackrel{def}{=} \bigwedge_{\vec{p}_i} \left(\bigvee_{\vec{b}_j} \vec{p}_i \approx \vec{b}_j \right) \wedge \bigwedge_{i \neq j} t_i \not\approx t_j.$$

Suppose each task t_i uses $band(t_i)$ percent of the bandwidth on the bus where it is placed. Then the sum of the utilizations on every bus must be less than 100:

$$\varphi_4 \stackrel{def}{=} \bigwedge util(\vec{b}_i, n) \leq 100.$$

The function $util(\vec{b}, j)$ produces a formula summing the bus utilization:

$$util(\vec{b}, j) \stackrel{def}{=} \begin{cases} ite(\vec{p}_1 \approx \vec{b}, band(\vec{t}_1), 0) & \text{if } j = 1, \\ ite(\vec{p}_j \approx \vec{b}, band(\vec{t}_j), 0) + util(\vec{b}, j-1) & \text{otherwise.} \end{cases}$$

The *if-then-else* expression $ite(\psi_1, \psi_2, \psi_3)$ is the usual shorthand for a fresh variable x and a side constraint $(\psi_1 \Rightarrow x \approx \psi_2) \wedge (\neg\psi_1 \Rightarrow x \approx \psi_3)$. Collecting together these constraints, the final constraint formula is:

$$\varphi \stackrel{def}{=} \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4.$$

The bottom of Figure 2 shows several complete architectures resulting from satisfactory decisions. For example, the leftmost architecture was produced by $decide(\theta)$ for:

$$\theta(\vec{t}_1) = a,\ \theta(\vec{t}_2) = b,\ \theta(\vec{b}_1) = \theta(\vec{p}_1) = 0,\ \theta(\vec{b}_2) = \theta(\vec{b}_3) = 1,$$
$$\theta(\vec{b}_4) = \theta(\vec{b}_5) = \theta(\vec{b}_6) = \theta(\vec{b}_7) = \theta(\vec{p}_2) = 2.$$

The solution space of this problem contains many isomorphic architectures, but these can be filtered out by fixing the equivalence relation to be graph isomorphism. The examples in the Figure 2 are all non-isomorphic.

4. SYMMETRY-DIRECTED RANDOMIZED PARTITIONING

We now describe the *Symmetry-Directed Randomized Partitioning* (SDRP) algorithm for reducing search bias on constrained graphs instances, where graph isomorphism is the equivalence relation $\sim$. SDRP does not require any modification to the underlying constraint solver. SDRP assumes constraints can be built from the usual boolean connectives $(\wedge, \vee, \neg)$ and equality $(\approx)$. The basic ideas of SDRP are:

- Force the solver into random regions of the solution space by asserting random *partitions* of variables.
- Use symmetries in the input problem to avoid jumping into isomorphic regions of the solution space.

- Provide a throttling mechanism that adjusts degree of randomization based on problem difficulty.

Through experimentation we found these three ingredients essential to reduce search bias without sacrificing performance.

4.1 Isomorphisms and Partitions

Before describing SDRP, we examine the problem of enumerating non-isomorphic solutions in more detail. Given a constrained graph instance $\mathcal{I}_H$ and a valuation θ_1 then there exists a formula $iso(\theta_1, H)$ that is satisfied by all isomorphic valuations θ':

$$\theta'(H) \sim \theta_1(H) \text{ if and only if } \theta' \models iso(\theta_1, H).$$

With this constraint in hand, a new non-isomorphic solution can be found by searching for a valuation θ_2 satisfying $\varphi \wedge \neg iso(\theta_1, H)$. This process can be repeated to enumerate the i^{th} non-isomorphic solution from $i-1$ solutions:

$$\theta_i \models \varphi \wedge \bigwedge_{1 \le j < i} \neg iso(\theta_j, H).$$

Unfortunately, the best techniques for computing $iso(\theta, H)$ have worst-case exponential time complexity and $iso(\theta, H)$ may be exponentially large [1, 2]. Thus, we shall work with simpler formulas called *partition constraints*.

Definition 8 (Partition)**.** A partition of H with vertices $V \stackrel{def}{=} \{\vec{y}_1, \ldots, \vec{y}_n\}$ is a set of non-empty disjoint sets $P \stackrel{def}{=} \{B_1, \ldots, B_m\}$ satisfying $V = B_1 \cup \ldots \cup B_m$. Each B_i is called a block.

Given the graph H in Figure 2, this partition places the root bus and its left and right subtrees into different blocks:

$$P_{ex} \stackrel{def}{=} \{\{\vec{b}_2, \vec{b}_4, \vec{b}_5\}, \{\vec{b}_1\}, \{\vec{b}_3, \vec{b}_6, \vec{b}_7\}, \{\vec{t}_1, \vec{t}_2, \vec{p}_1, \vec{p}_2\}\}.$$

Alternatively, a partition can be phrased as a constraint among vertex labels.

Definition 9 (Partition Constraint)**.** The partition constraint of P is a formula $\psi(P) \stackrel{def}{=} \psi_{\approx}(P) \wedge \psi_{\not\approx}(P)$ where:

$$\psi_{\approx}(P) \stackrel{def}{=} \bigwedge_{\vec{y}, \vec{z} \in B_i} \vec{y} \approx \vec{z}. \quad \psi_{\not\approx}(P) \stackrel{def}{=} \bigwedge_{\substack{\vec{y} \in B_i, \vec{z} \in B_j \\ \text{and } i \neq j}} \vec{y} \not\approx \vec{z}.$$

For example,

$$\psi(P_{ex}) = \begin{array}{l} \vec{b_2} \approx \vec{b_4} \approx \vec{b_5} \wedge \vec{b_3} \approx \vec{b_6} \approx \vec{b_7} \wedge \\ \vec{t_1} \approx \vec{t_2} \approx \vec{p_1} \approx \vec{p_2} \wedge \\ \vec{b_1} \not\approx \vec{b_2} \wedge \ \vec{b_1} \not\approx \vec{b_3} \wedge \vec{b_1} \not\approx \vec{b_4} \wedge \ \vec{b_1} \not\approx \vec{b_5} \wedge \ldots. \end{array}$$

Notice the partition constraint has no more than $O(|H|^2)$ conjuncts, so it can be computed efficiently.

Every valuation θ can be generalized to a partition called the *kernel partition*.

Definition 10 (Kernel Partition)**.** The kernel partition of θ is a partition $Ker(\theta)$ such that $\vec{y}$ and $\vec{z}$ are in the same block if and only if $\theta(\vec{y}) = \theta(\vec{z})$.

Recall the earlier valuation:

$$\theta(\vec{t}_1) = a,\ \theta(\vec{t}_2) = b,\ \theta(\vec{b}_1) = \theta(\vec{p}_1) = 0,\ \theta(\vec{b}_2) = \theta(\vec{b}_3) = 1,$$
$$\theta(\vec{b}_4) = \theta(\vec{b}_5) = \theta(\vec{b}_6) = \theta(\vec{b}_7) = \theta(\vec{p}_2) = 2.$$

Its kernel partition is:

$$Ker(\theta) = \{\{\vec{t}_1\}, \{\vec{t}_2\}, \{\vec{b}_1, \vec{p}_1\}, \{\vec{b}_2, \vec{b}_3\}, \{\vec{b}_4, \vec{b}_5, \vec{b}_6, \vec{b}_7, \vec{p}_2\}\}$$

Combining these ideas leads to an polynomial-sized approximation of isomorphism classes:

Lemma 5. *If $\theta' \models \psi(Ker(\theta))$ then $\theta' \models iso(\theta, H)$.*

Proof. We must show that whenever two valuations θ and θ' have the same kernel, then the graphs $\theta(H)$ and $\theta'(H)$ are isomorphic. Recall, $\theta(H)$ and $\theta'(H)$ are isomorphic if there is a bijection f between their vertices such that (u, v) is an edge in $\theta(H)$ if and only if $(f(u), f(v))$ is an edge in $\theta'(H)$.

Assume $\theta' \models \psi(Ker(\theta))$ and define the function $f(\theta(\vec{y})) \stackrel{def}{=} \theta'(\vec{y})$. If f is not well-defined then there exists $\vec{y}$ and $\vec{z}$ such that $\theta(\vec{y}) = \theta(\vec{z})$ but $\theta'(\vec{y}) \neq \theta'(\vec{z})$. This is impossible because θ' has the same kernel as θ. By construction, f is onto. Assume it is not one-to-one, then there exists $\vec{y}$ and $\vec{z}$ such that $\theta(\vec{y}) \neq \theta(\vec{z})$ but $\theta'(\vec{y}) = \theta'(\vec{z})$. Again this is impossible because θ has the same kernel as θ'. Thus, f is a bijection.

Suppose $(\theta(\vec{y}), \theta(\vec{z}))$ is an edge of $\theta(H)$, then $(\vec{y}, \vec{z})$ is an edge of H and $(\theta'(\vec{y}), \theta'(\vec{z}))$ is an edge of $\theta'(H)$. Applying the function f to this edge:

$$f(\theta(\vec{y}), \theta(\vec{z})) = (f(\theta(\vec{y})), f(\theta(\vec{z}))) = (\theta'(\vec{y}), \theta'(\vec{z})).$$

Thus, f is homomorphism from $\theta(H)$ to $\theta'(H)$. Applying the same argument for f^{-1} establishes that f is an isomorphism. □

Using Lemma 5, the i^{th} solution is approximated by blocking the previous $i-1$ kernel partitions:

$$\theta_i \models \varphi \wedge \bigwedge_{1 \le j < i} \neg \psi(Ker(\theta_j)).$$

In the worst case, this procedure may return an exponential number of solutions before leaving the isomorphism class.

4.2 Symmetries and Randomization

The partition blocking algorithm ensures that all isomorphism classes are eventually visited, but it does not reduce bias introduced by the size of the isomorphism classes and the intrinsic bias of the solver. We shall augment this algorithm by asserting random partitions, forcing the solver to visit new equivalence classes:

$$\theta_i \models \varphi \wedge \psi(P_{rand}) \wedge \bigwedge_{1 \le j < i} \neg \psi(Ker(\theta_j)).$$

where P_{rand} is a random partition not currently blocked.

Because partitions approximate isomorphism classes, a random partition might force the solver back into an old isomorphism class or into an unsatisfiable region of the space. We address the first problem by recognizing that the initial graph H contains information about the isomorphism classes. This information shall be extracted and used to direct the randomization, thereby decreasing the probability that a random partition describes an old isomorphism class.

Definition 11 (Automorphism)**.** An automorphism α of $\mathcal{I}_H$ is a bijection from variables $\alpha : \vec{x} \to \vec{x}$ satisfying $\alpha(H) = H$ and $\theta \models \varphi[\vec{x}]$ iff $(\theta \circ \alpha) \models \varphi[\vec{x}]$. Let $aut(\mathcal{I}_H)$ be the set of all such automorphisms.

As before, $aut(\mathcal{I}_H)$ may be exponentially large, but there is a polynomial-sized subset $gen(\mathcal{I}_H) \subseteq aut(\mathcal{I}_H)$ that generates all automorphisms through sequential composition. This set is called the *generator set of the automorphism group* of $\mathcal{I}_H$. State-of-the-art techniques for isomorphism detection and *symmetry breaking* efficiently compute the generators of various automorphism groups [1]. For example, the generators for H in Figure 2 are:

$$\alpha_1(\vec{x}) \stackrel{def}{=} \begin{cases} \vec{t}_1 & \text{if } \vec{x} = \vec{t}_2, \\ \vec{t}_2 & \text{if } \vec{x} = \vec{t}_1, \\ \vec{p}_1 & \text{if } \vec{x} = \vec{p}_2, \\ \vec{p}_2 & \text{if } \vec{x} = \vec{p}_1, \\ \vec{x} & \text{otherwise.} \end{cases} \quad \alpha_2(\vec{x}) \stackrel{def}{=} \begin{cases} \vec{b}_4 & \text{if } \vec{x} = \vec{b}_5, \\ \vec{b}_5 & \text{if } \vec{x} = \vec{b}_4, \\ \vec{x} & \text{otherwise.} \end{cases}$$

$$\alpha_3(\vec{x}) \stackrel{def}{=} \begin{cases} \vec{b}_6 & \text{if } \vec{x} = \vec{b}_7, \\ \vec{b}_7 & \text{if } \vec{x} = \vec{b}_6, \\ \vec{x} & \text{otherwise.} \end{cases}$$

$\alpha_4(\vec{b}_2) \stackrel{def}{=} \vec{b}_3$, $\alpha_4(\vec{b}_3) \stackrel{def}{=} \vec{b}_2$,
$\alpha_4(\vec{x}) \stackrel{def}{=} \alpha_2(\alpha_3(\vec{x}))$ if $\vec{x} \notin \{\vec{b}_2, \vec{b}_3\}$.

The generators reveal the symmetries of the input problem and will be used rewrite random partitions. If a random partition P is rewritten into a blocked partition P', then P is just a different name for the isomorphism class named by P'. In this case P can be ignored and another partition guessed. Let $\alpha(P)$ be the partition formed by applying α to every element in every block of P.

Lemma 6. *If $P' = \alpha(P)$ then all valuations satisfying $\psi(P)$ and all valuations satisfying $\psi(P')$ are in the same isomorphism class.*

Proof. Pick any $\theta \models \psi(P)$ and any $\theta' \models \psi(\alpha(P))$. Define $f(\theta(\vec{y})) \stackrel{def}{=} \theta'(\alpha(\vec{y}))$. Following the logic of Lemma 5, f must be a bijection otherwise α is not a bijection or θ' does satisfy $\psi(\alpha(P))$.

Suppose $(\theta(\vec{y}), \theta(\vec{z}))$ is an edge of $\theta(H)$, then $(\vec{y}, \vec{z})$ and $(\alpha(\vec{y}), \alpha(\vec{z}))$ are edges of H and $(\theta'(\alpha(\vec{y})), \theta'(\alpha(\vec{z})))$ is an edge of $\theta'(H)$. Applying the function f to this edge:

$$f(\theta(\vec{y}), \theta(\vec{z})) = (f(\theta(\vec{y})), f(\theta(\vec{z}))) = (\theta'(\alpha(\vec{y})), \theta'(\alpha(\vec{z}))).$$

Thus, f is homomorphism from $\theta(H)$ to $\theta'(H)$. Applying the same argument for f^{-1} and α^{-1} establishes that f is an isomorphism. □

4.3 The Algorithm

SDRP uses $gen(\mathcal{I}_H)$ to avoid redundant random partitions. However, Lemma 6 requires searching for the automorphism α witnessing the redundancy of P. We implement this search by a minimization procedure that repeatedly rewrites partitions into minimal partitions via generators. If two partitions have the same minimum partition then α exists. First, observe that every partition can be uniquely labeled by the bit vector $label(P)$.

Definition 12 (Partition Label)**.** Every P can be described by a bit-vector containing $|H|^2$ bits.

$$label(P) \stackrel{def}{=} b_{1,1}\ b_{1,2}\ \ldots\ b_{1,n}\ b_{2,2}\ b_{2,3}\ \ldots\ b_{n-1,n}\ b_{n,n},$$

where $b_{i,j} \stackrel{def}{=} 1$ if $\vec{y}_i$ and $\vec{y}_j$ are in the same block B.

Partition labels are ordered by the usual total order $\leq$ on bit vectors.

Corollary (Automorphism Witness)**.** *There exists an $\alpha \in aut(\mathcal{I}_H)$ such that $P' = \alpha(P)$ if and only if*

$$\min_{\alpha' \in aut(\mathcal{I}_H)} \{label(\alpha'(P))\} = \min_{\alpha' \in aut(\mathcal{I}_H)} \{label(\alpha'(P'))\}.$$

Algorithm 2 SDRP($\mathcal{I}_H$, m, c_0)

```
 1: b ← {}        // Set of blocked partitions
 2: s ← {}        // Set of solutions
 3: c ← c_0       // Initial value of PDF parameter
 4: φ_enum ← φ // Initial constraint formula
 5: gen ← ComputeGen(I_H)
 6: while |s| < m do
 7:   repeat
 8:     P ← DrawPartition(H, c)
 9:     P_min ← GreedyMinimize(P, gen)
10:   until ¬IsBlocked(b, P_min)
11:   if IsSat(φ_enum ∧ ψ≈(P_min)) then
12:     θ ← Solve(φ_enum ∧ ψ≈(P_min))
13:     s ← s ∪ {θ(H)}
14:     φ_enum ← φ_enum ∧ ¬ψ(Ker(θ))
15:     c ← c + ϵ
16:   else if |P_min| < |H| then
17:     b ← b ∪ {P_min}
18:     c ← c − ϵ
19:   else
20:     return s
21:   end if
22: end while
23: return s
```

Minimization is implemented by repeatedly applying generators to labels. This allows the search effort to be tuned. For instance, search can be terminated after finding a locally minimal label as opposed to the globally minimum label. It also simplifies data structures; the key data structure is a database of minimized partition labels.

We are now ready to present the SDRP algorithm. Its input is a constrained graph $\mathcal{I}_H$, an integer $m \geq 0$ and an initial condition c_0. It assumes a solver provides the methods **IsSat** and **Solve**. **Solve** returns a valuation whereas **IsSat** only returns true/false. The algorithm returns m solutions from distinct partition classes, or as many as can be found if there are fewer than m satisfiable partition classes. The m solutions are not guaranteed to be non-isomorphic, because isomorphism testing is approximated by greedy minimization of partitions.

The algorithm keeps track of the search state through four variables. The solution set s contains the current set of solutions. The constraint formula φ_{enum} is the current constraint based on previously enumerated solutions; it is strengthened as more solutions are enumerated. The variable c adjusts how partitions are randomly selected. Finally, the blocked partition set b contains a partition P if it is known that:

- The equality fragment of the partition constraint, $\psi_{\approx}(P)$, is unsatisfiable, and
- Every partition P', for which $\psi_{\approx}(P')$ implies $\psi_{\approx}(P)$, is unsatisfiable.

Intuitively, if P is in the blocked partition set, then all partitions with at least the same equalities as P must also be unsatisfiable. Initially the constraint formula is just φ and the sets s, b are empty.

The algorithm repeatedly generates and minimizes random partitions (Lines 7 - 10). The **DrawPartition** method implements a family of probability distributions on partitions with a single parameter c. The larger the value of this

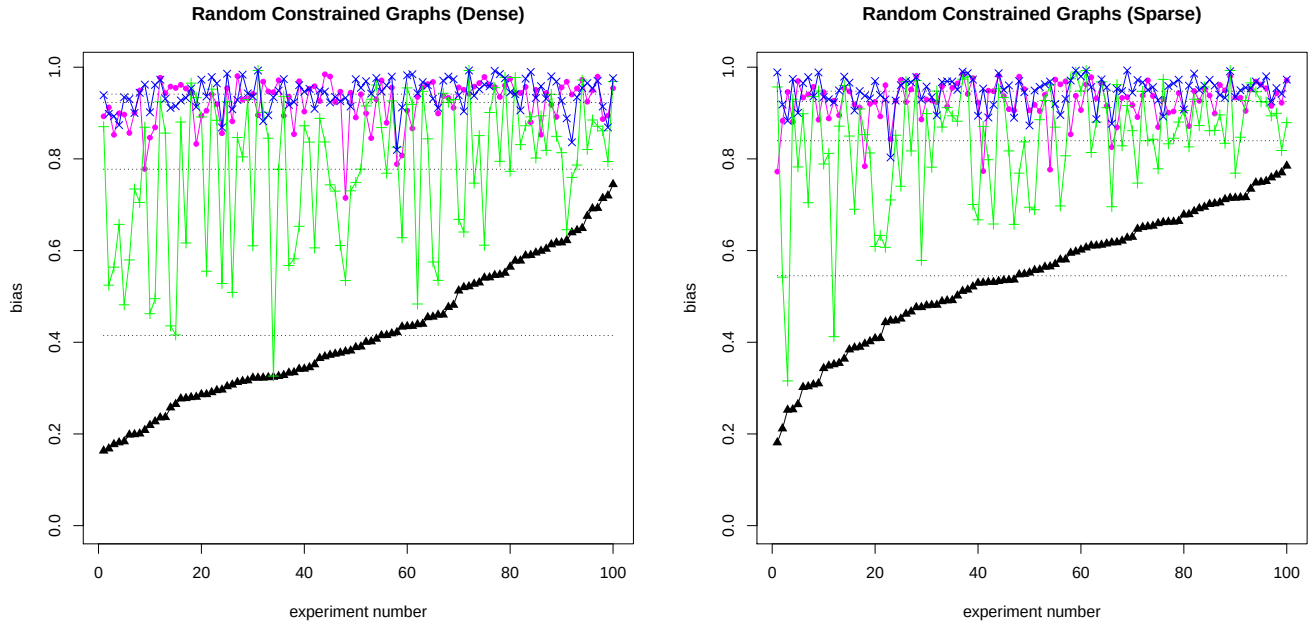

Figure 3: Random Graphs; SDRP with Z3 (▲), LIB with Z3 (•), LIB with OpenSMT (×), LIB-RS with OpenSMT (+).

parameter, the higher the probability of drawing partitions with larger blocks; its initial condition is c_0. **GreedyMinimize** minimizes a partition P by repeatedly applying generators. These methods can be implemented in different ways. In our implementation, **DrawPartition** exponentially favors small block sizes and **GreedyMinimize** employs a beam search. The blocked partition set b remembers unsatisfiable partitions. If a random partition is blocked, then it is discarded and another is chosen (Line 10).

Once a feasible partition is generated, it is checked for satisfiability against the current constraint formula (Line 11). Importantly, only the equality fragment of the partition is enforced. This allows the solver to search over many partitions with a single call, and provides a simple termination condition for the algorithm. If $\varphi_{enum} \wedge \psi_{\approx}(P_{min})$ is satisfiable, then a satisfying valuation θ is extracted and s is updated (Lines 12 - 13). Next, the constraint formula is updated to block the kernel partition of θ. At this point, no valuation with the same kernel partition as θ can be enumerated. Finally, the parameter c is updated to increase the likelihood of partitions with larger blocks (harder partitions).

If the equality fragment of the random partition is unsatisfiable, then all partitions with at least the same equalities as P_{min} are unsatisfiable. This scheme allows the algorithm to generalize one instance of unsatisfiability to many partitions. If P_{min} contains only trivial equalities, then it has $|H|$ blocks and all remaining partitions are unsatisfiable. In this case the algorithm terminates (Line 20). Otherwise, the partition is added to b and the parameter c is decremented to favor easier partitions (Lines 16 -18).

In summary, the SDRP algorithm uses the parameter c to throttle the difficulty of random partitions. It constrains the solver with $\psi_{\approx}$ to search over many partitions at once, and to generalize unsatisfiability to many partitions. We found these ingredients crucial for introducing randomization without over-constraining the search problem. We now present experimental results.

5. EXPERIMENTS

We now compare several enumeration algorithms on constrained graph problems by inferring urn models using Algorithm 1 and then computing bias. The backend solvers selected for comparison were state-of-the-art *satisfiability modulo theories* (SMT) solvers, which are now routinely used to solve hard enumeration problems in software engineering. We limited our comparisons to two solvers, *Z3* [5] and *OpenSMT* [3], based on a common input language and ease-of-modification of the solvers' source. Z3 is currently the fastest industrial SMT solver. We tested it on its own and as the backend solver to our SDRP algorithm. OpenSMT is an open source SMT solver. Its search engine is based on the popular *MiniSAT* solver [7], so evaluation of OpenSMT on purely boolean constraints gives a reasonable approximation of MiniSAT's search behavior. Because OpenSMT is open source, we also developed a modified version of OpenSMT that maximizes randomization within the solver's kernel. This allows us to test the affects of low level randomization on search bias.

Using these solvers we developed two additional enumeration algorithms:

1. **Lazy isomorphism blocking** (LIB): This algorithm maintains a database of non-isomorphic solutions. It draws solutions from the solver until a new non-isomorphic solution is discovered and then returns this solution. Each time a solution is found that specific solution is blocked. LIB tests the baseline affects of isomorphism classes on search bias.

2. **LIB with randomized solver** (LIB-RS): The LIB algorithm with a modified solver that makes random search decisions with maximum probability. This algorithm tests if increasing randomization in the solver's core reduces search bias.

Testing enumerations algorithms required benchmark problems where we could count the number of satisfiable isomorphism classes and correctly label solutions by isomorphism classes. Due to the combinatorics of search problems this limits the size of benchmark problems. Our *random graph* benchmarks were problem instances made of a random graph and a random hard boolean constraint. The graph component was an *Erdos-Renyi* random graph of 15 vertices, 60 propositional variables, and four variables per vertex label. The hard constraint component was generated by randomly generating 3-CNF problems with a ratio of clauses to variables around the phase transition of 3-CNF [6]. The resulting problem instances were small enough to allow for explicit enumeration (via careful implementation), but still large enough to exercise the solvers. The first set of random graph benchmarks were on dense random graphs, where there are many small isomorphism classes. These problem instances had 5,000 to 20,000 equivalence classes. The second set of random benchmarks were on sparse random graphs, which have a few thousand large isomorphism classes. On average it was harder to eliminate bias from the sparse case.

The third benchmark consisted of enumerating bus topologies according to Example 1. Each problem instance had 13 bus / placement variables, 10 task variables, and a set of random bandwidth requirements. Typical random problem instances were highly constrained. Up to several hundred thousand isomorphism classes could exist for easy bandwidth requirements, but small changes in bandwidth requirements could shrink the solution space to several dozen classes. Unlike the sparse graph instances, the equivalence classes were both few in number and contained few solutions. The baseline algorithms showed less bias for these instances. Our hypothesis is that highly constrained problem instances resulted in significant backtracking within the solver thereby enhancing diversity. This benchmark also tests the integer arithmetic subsolvers of Z3 and OpenSMT.

Each benchmark consisted of 100 random experiments (problem instances). Each experiment consisted of drawing 20 non-isomorphic solutions and repeating this 50 times with different random seeds. After 50 repetitions an urn model was built for the experiment and bias estimated. These benchmarks took several weeks to run on a 3.16 GHz Intel Core 2 Duo with 4 GB of memory; Figures 3 and 4 show the results. In order to improve readability, the experiments were *a posteriori* numbered in ascending order using their SDRP bias. The y axis shows the estimated bias $\hat{\beta}$ for each experiment. Grey lines show the average bias for each algorithm on the benchmark.

The results show solvers are highly biased in their search. They return a few solutions with very high probability, regardless of their initial conditions and with mild sensitivity to the structure of equivalence classes. Increasing randomization in the solver's core yields some improvement, but not dramatically so. Also, this approach has negative side effects on the solver's performance. The SDRP algorithm significantly reduces solver bias using detailed information about the symmetries present in problems. It has the best average behavior and the best absolute behavior on 97.6% of all experiments.

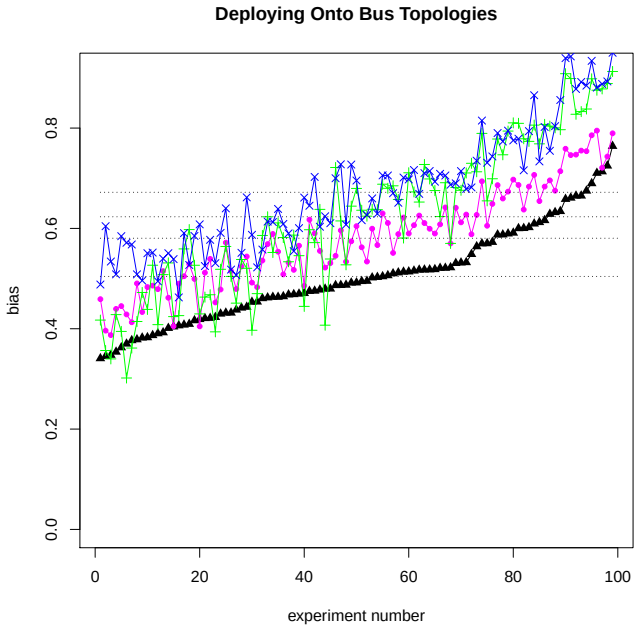

Figure 4: Bus topologies; Same legend as Figure 3.

6. RELATED WORK

At first glance, many synthesis problems appear to be optimization problems. Theoretically, there is a best solution; examining multiple candidates should be unnecessary. In practice, competing optimization goals (energy vs. speed, throughput vs. cost, etc...) correspond to *multi-criteria optimization problems* [16, 10]. These problems result in a *Pareto front* of non-dominated solutions that must be explored. Recent approaches to diversity incorporate solution-wise distance measures into evolutionary search. The enumerator should return a subset of the Pareto front where elements are maximally distant [22].

The work of [16] shows that optimization problems can be approximated with constraint solvers. Similarly, distance measures can also be pushed into constraint solvers using easily defined distance measures. For example, the work of [18] uses Hamming distance over an ordering of propositional variables. The solver then repeatedly queries for solutions with large Hamming distances from previous solutions. Other approaches force the solver to frequently make random decisions. We experimentally showed that this approach improves diversity somewhat.

Distance-based definitions of diversity require selecting a meaningful measure on the solution space. For example, Hamming distance is simple to define, but it may not be meaningful. Our examples would require a distance measure that closely groups isomorphic solutions; it is certainly non-trivial to construct such a measure. We dispense with distances and opt for random sampling over equivalence classes. Our definition of diversity is w.r.t. the enumeration process, i.e. its proximity to a uniform sampler of equivalence classes.

Diversity is becoming increasingly important in the field of information retrieval. Given a query, the goal is to find a set of highly relevant documents matching the query. In this field, diversity is a measure of the number of documents that can be included in the results without significantly reducing the relevance of the results [23, 4]. Though similar ideas could be useful for exploring solution spaces in embedded systems, it is unclear how these algorithms can be applied to solution spaces that are implicitly defined by hard constraints.

7. CONCLUSION AND FUTURE WORK

We presented a novel theory to model and measure diversity in enumeration algorithms using Wallenius Urns. Next, we developed a broad class of enumeration problems, called constrained graphs, that unify many enumeration problems. We presented the SDRP algorithm for increasing diversity when enumerating solutions to constrained graphs. Finally, we reported experimental results showing that SDRP significantly reduces search bias on several benchmarks. This algorithm has been implemented in the FORMULA framework using the state-of-the-art SMT solver Z3.

Extensions to this work include development of more benchmarks with other types of constraints and subsolvers. More generally, we would like to understand how diversity correlates with problem structure. This would be useful to throttle generation of partition constraints and to notify the engineer early about the expected difficulty of extracting diverse solutions. Finally, we plan to study a deeper integration of diverse enumerators with optimization techniques.

We believe that our approach to diverse enumeration synergizes with the types of (evolutionary) search techniques commonly employed for non-linear optimization, e.g. for the purposes of generating optimized mappings and schedules [10]. Evolutionary techniques benefit from smooth fitness landscapes, but use biologically inspired techniques to jump across poorly performing regions of the space. On the other hand, constraint solvers typically do not incorporate fitness functions, but are a proven technology for solution spaces where most choices are maximally bad (i.e. unsatisfiable). A diverse enumerator may be useful for sampling and seeding other search algorithms with initially satisfiable point designs for spaces governed by difficult constraints.

8. REFERENCES

[1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *DAC*, pages 836–839, 2003.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[3] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *TACAS*, pages 150–153, 2010.

[4] C. L. A. Clarke, M. Kolla, G. V. Cormack, O. Vechtomova, A. Ashkan, S. Büttcher, and I. MacKinnon. Novelty and diversity in information retrieval evaluation. In *SIGIR*, pages 659–666, 2008.

[5] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[6] O. Dubois. Upper bounds on the satisfiability threshold. *Theor. Comput. Sci.*, 265(1-2):187–197, 2001.

[7] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[8] S. Fischmeister, O. Sokolsky, and I. Lee. A verifiable language for programming real-time communication schedules. *IEEE Trans. Computers*, 56(11):1505–1519, 2007.

[9] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In *EMSOFT*, pages 63–72, 2012.

[10] A. Hamann and R. Ernst. Tdma time slot and turn optimization with evolutionary search techniques. In *DATE*, pages 312–317, 2005.

[11] C. Hang, P. Manolios, and V. Papavasileiou. Synthesizing cyber-physical architectural models with real-time constraints. In *CAV*, pages 441–456, 2011.

[12] T. A. Henzinger, C. M. Kirsch, E. R. B. Marques, and A. Sokolova. Distributed, modular htl. In *RTSS*, pages 171–180, 2009.

[13] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, Platforms and Possibilities: Towards Generic Automation for MDA. In *EMSOFT*, pages 39–48, 2010.

[14] N. Johnson, S. Kotz, and N. Balakrishnan. *Discrete Multivariate Distributions*. Wiley-Interscience, 1996.

[15] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *LICS*, pages 389–398, 2001.

[16] J. Legriel, C. L. Guernic, S. Cotton, and O. Maler. Approximating the pareto front of multi-criteria optimization problems. In *TACAS*, pages 69–83, 2010.

[17] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *SPLC*, pages 136–150, 2010.

[18] A. Nadel. Generating Diverse Solutions in SAT. In *SAT*, pages 287–301, 2011.

[19] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[20] F. Reimann, M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich. Symbolic system synthesis in the presence of stringent real-time constraints. In *DAC*, pages 393–398, 2011.

[21] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647, 2007.

[22] T. Ulrich and L. Thiele. Maximizing population diversity in single-objective optimization. In *GECCO*, pages 641–648, 2011.

[23] D. Vallet and P. Castells. Personalized diversification of search results. In *SIGIR*, pages 841–850, 2012.

[24] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand, and P. L. Guernic. Polychronous controller synthesis from marte ccsl timing specifications. In *MEMOCODE*, pages 21–30, 2011.