Scalable Techniques for Constrained Sampling and Counting

Kuldeep S. Meel
Rice University

Joint work with Supratik Chakraborty (IITB), Daniel J. Fremont (UCB), Alexander Ivrii (IBM), Sharad Malik (Princeton), Sanjit A. Seshia (UCB), Moshe Y. Vardi (Rice)
How do we guarantee that systems work **correctly**?

**Functional Verification**

- Formal verification
  - Challenges: formal requirements, scalability
  - ~10-15% of verification effort
- Dynamic verification: *dominant approach*
Dynamic Verification

- Design is simulated with test vectors
  - Test vectors represent different verification scenarios
- Results from simulation compared to intended results
- **Challenge**: Exceedingly large test space!
Motivating Example

How do we test the circuit works?

- Try for all values of $a$ and $b$
- $2^{128}$ possibilities
- Sun will go nova before done!
- Not scalable
Constrained-Random Simulation

Sources for Constraints

- Designers:
  1. \(a +_{64} 11 \times_{32} b = 12\)
  2. \(a <_{64} (b >> 4)\)
- Past Experience:
  1. \(40 <_{64} 34 + a <_{64} 5050\)
  2. \(120 <_{64} b <_{64} 230\)
- Users:
  1. \(232 \times_{32} a + b != 1100\)
  2. \(1020 <_{64} (b /_{64} 2) +_{64} a <_{64} 2200\)

- Test vectors: solutions of constraints
Constrained-Random Simulation

Sources for Constraints

- Designers:
  1. \(a +_{64} 11 \times_{32} b = 12\)
  2. \(a <_{64} (b >> 4)\)

- Past Experience:
  1. \(40 <_{64} 34 + a <_{64} 5050\)
  2. \(120 <_{64} b <_{64} 230\)

- Users:
  1. \(232 \times_{32} a + b \neq 1100\)
  2. \(1020 <_{64} (b /_{64} 2) +_{64} a <_{64} 2200\)

Problem: How can we uniformly sample the values of \(a\) and \(b\) satisfying the above constraints?
Problem Formulation

Scalable Uniform Generation of SAT Witnesses

\[ c = f(a, b) \]

Set of Constraints

SAT Formula

Sample satisfying assignments uniformly at random
Constraint satisfaction for random stimuli generation

Yehuda Naveh
IBM Haifa Research Lab
Diverse Applications

Search-based Synthesis

Constrained Random Simulation

SAT Sampling

Probabilistic Inference

Planning under uncertainty

Automatic Problem Generation
Search-Based Synthesis

- **Goal**: synthesize from under-constrained specifications (“sketch”)
- Large space of programs that satisfy correctness conditions
- Task: Find “optimal” program (wrt running time, memory, …)
- Method: *Uniformly sample* from the space of programs
Constrained Counting

• Given a SAT formula F
• $R_F$: Set of all solutions of F
• Problem (#SAT): Estimate the number of solutions of F ($#F$) i.e., what is the cardinality of $R_F$?
• E.g., $F = (a \lor b)$
• $R_F = \{(0,1), (1,0), (1,1)\}$
• The number of solutions ($#F$) = 3

#P: The class of counting problems for decision problems in NP!
Practical Applications

Wide range of applications!

- Probabilistic reasoning/Bayesian inference
- Dynamic Verification
- Planning with uncertainty
- Multi-agent/ adversarial reasoning

[Roth 96, Sang 04, Bacchus 04, Domshlak 07]
Agenda

Design **Scalable** Techniques for Uniform Generation and Model Counting with **Strong** Theoretical Guarantees
Agenda

Design **Scalable** Techniques for Almost-Uniform Generation and Approximate-Model Counting with **Strong Theoretical Guarantees**
Prior Work

Guarantees

Performance

- BGP
- BDD

- MCMC
- SAT-Based
## Desires

<table>
<thead>
<tr>
<th>Generator</th>
<th>Relative runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-of-the-art: XORSample’</td>
<td>50000</td>
</tr>
<tr>
<td>Ideal Uniform Generator*</td>
<td>10</td>
</tr>
<tr>
<td>SAT Solver</td>
<td>1</td>
</tr>
</tbody>
</table>

Experiments over 200+ benchmarks
*: According to EDA experts
Our Contribution

Performance

Guarantees

BGP  BDD

UniGen

MCMC  SAT-Based
Partitioning into equal “small” cells
Partitioning into equal “small” cells
Partitioning into equal “small” cells

- Pick a random cell
- Pick a random solution from this cell
How to Partition?

How to partition into roughly equal small cells of solutions without knowing the distribution of solutions?

Universal Hashing
[Carter-Wegman 1979] (IBM Research)
Universal Hashing

• Hash functions: mapping \(\{0,1\}^n\) to \(\{0,1\}^m\)
  • (\(2^n\) elements to \(2^m\) cells)

• Random inputs => All cells are \textit{roughly} equal (in expectation)

• Universal family of hash functions:
  • Choose hash function randomly from family
  • For \textit{arbitrary} distribution on inputs => All cells are \textit{roughly equal} (in expectation)
Universal Hashing and Independence

• Hash functions from mapping \( \{0,1\}^n \) to \( \{0,1\}^m \)
  - \( (2^n \text{ elements to } 2^m \text{ cells}) \)

• Universal hash functions:
  • Choose hash function randomly
  • For arbitrary distribution on inputs => All cells are *roughly* equal in expectation
  • But:
    • While each input is hashed *uniformly*
    • Different inputs *might not* be hashed *independently*
Strong Universality

- $H(n,m,r)$: Family of $r$-universal hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$ ($2^n$ elements to $2^m$ cells)
  - $r$: degree of independence of hashed inputs

- Higher $r$ => Stronger guarantee on range of size of cells

- $r$-wise universality => Polynomials of degree $r-1$

- Higher universality => Higher complexity
Partitioning

• How large should the cells be?

• How many cells?
Size of cell

- Too large => Hard to enumerate
- Too small => Variance can be very high

\[ \text{pivot} = 5\left(1 + \frac{1}{\varepsilon}\right)^2 \]
How many cells?

• Our desire:  \( 2^m = \frac{|R_F|}{\text{pivot}} \)
  
  • But determining \(|R_F|\) is expensive (#P complete)

• How about approximation?
  
  • \textit{ApproxMC} \((F, \varepsilon, \delta)\) returns \(C\):
    \[
    \Pr\left[ \frac{|R_F|}{1+\varepsilon} \leq C \leq (1 + \varepsilon)|R_F| \right] \geq 1 - \delta
    \]
  
  • \(q = \log C - \log \text{pivot}\)

  • Concentrate on \(2^m\), where \(m = q-1, q, q+1\)
ApproxMC\((F, \varepsilon, \delta)\)

- For right choice of \(m\), large number of cells are “small”
- “almost all” the cells are “roughly” equal
- Check if a randomly picked cell is “small”
- If yes, then estimate = # of solutions in cell * \(2^m\)
ApproxMC(F, ε, δ)

#sols < pivot

NO
ApproxMC(F, \varepsilon, \delta)

#sols < pivot

NO
ApproxMC(F, ε, δ)

Estimate:
# of sols * 2^m
Runtime Performance of ApproxMC
Can Solve a Large Class of Problems

Large class of problems that lie beyond the exact counters but can be computed by ApproxMC
Mean Error: Only 4% (allowed: 75%)

Mean error: 4% – much smaller than the theoretical guarantee of 75%
Guarantees and Runtime performance of UniGen
Strong Theoretical Guarantees

- **Almost-Uniformity**

  For every solution $y$ of $R_F$

  \[
  \frac{1}{(6.84+\varepsilon) \times |R_F|} \leq \Pr\{y \text{ is output}\} \leq \frac{(6.84+\varepsilon)}{|R_F|}
  \]

- **Success Probability**

  UniGen succeeds with probability at least 0.52

  - In practice, succ. Probability $\sim 0.99$

- **Polynomial number of calls to SAT Solver**
1-2 Orders of Magnitude Faster

![Graph showing comparison between UniGen and XORSample'] (UniGen vs XORSample')
Results: Uniformity

- Benchmark: case110.cnf; \#var: 287; \#clauses: 1263
- Total Runs: $4 \times 10^6$; Total Solutions: 16384
Results: Uniformity

- Benchmark: case110.cnf; \#var: 287; \#clauses: 1263
- Total Runs: $4 \times 10^6$; Total Solutions: 16384
So far

- The first scalable approximate model counter
- The first scalable uniform generator
- Outperforms state-of-the-art generators/counters

Are we done?
## Where are we?

<table>
<thead>
<tr>
<th>Generator</th>
<th>Relative runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-of-the-art: XORSample’</td>
<td>50000</td>
</tr>
<tr>
<td>UniGen</td>
<td>~5000</td>
</tr>
<tr>
<td>Ideal Uniform Generator*</td>
<td>10</td>
</tr>
<tr>
<td>SAT Solver</td>
<td>1</td>
</tr>
</tbody>
</table>

Experiments over 200+ benchmarks

*: According to EDA experts
XOR-Based Hashing

- Partition $2^n$ space into $2^m$ cells
- Variables: $X_1, X_2, X_3, \ldots, X_n$
- Pick every variable with prob. $\frac{1}{2}$, XOR them and add $0/1$ with prob. $\frac{1}{2}$
- $X_1 + X_3 + X_6 + \ldots + X_{n-1} + 0$
- To construct $h$: $\{0,1\}^n \rightarrow \{0,1\}^m$, choose $m$ random XORs
- $\alpha \in \{0,1\}^m \rightarrow$ Set every XOR equation to 0 or 1 randomly
- The cell: $F \land$ XOR (CNF+XOR)
XOR-Based Hashing

• CryptoMiniSAT: Efficient for CNF+XOR

• Avg Length : n/2

• Smaller XORs ➔ better performance

How to shorten XOR clauses?
**Independent Support**

- Set I of variables such that assignments to these uniquely determine assignments to rest of variables (for satisfying assignments)

- If $\sigma_1$ and $\sigma_2$ agree on I then $\sigma_1 = \sigma_2$

- $c \leftarrow (a \lor b)$ ; Independent Support I: \{a, b\}

- **Key Idea**: Hash only on the independent variables

- Average size of XOR: $n/2$ to $|I|/2$
Key Idea

Minimal Independent Support (MIS)  \rightarrow  Minimal Unsatisfiable Subset (MUS)
Minimal Unsatisfiable Subset

- Given $\Psi = H_1 \land H_2 \cdots H_m$

  - Find subset $\{H_{i_1}, H_{i_2}, \cdots H_{i_k}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i_1} \land H_{i_2} \cdots H_{i_k} \land \Omega$ is UnsAT

  Unsatisfiable subset

- Find **minimal** subset $\{H_{i_1}, H_{i_2}, \cdots H_{i_k}\}$ of $\{H_1, H_2, \cdots H_m\}$ such that $H_{i_1} \land H_{i_2} \cdots H_{i_k}$ is UnsAT

  Minimal Unsatisfiable subset
Impact on Sampling and Counting Techniques
What about complexity

- Computation of MUS: $F^P^{NP}$

- Why solve a $F^P^{NP}$ for almost-uniform generation/approximate counter (PTIME PTM with NP Oracle)

Settling the debate through practice!
Performance Impact on Uniform Sampling
Where are we?

<table>
<thead>
<tr>
<th>Generator</th>
<th>Relative runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-of-the-art: XORSample’</td>
<td>50000</td>
</tr>
<tr>
<td>UniGen</td>
<td>5000</td>
</tr>
<tr>
<td>UniGen1</td>
<td>470</td>
</tr>
<tr>
<td>Ideal Uniform Generator*</td>
<td>10</td>
</tr>
<tr>
<td>SAT Solver</td>
<td>1</td>
</tr>
</tbody>
</table>
Back to basics

# of solutions in “small” cell $\in [\text{loThresh}, \text{hiThresh}]$
We pick one solution
“Wastage” of loThresh solutions

Pick $\text{loThresh}$ samples!
Balancing Independence

For $h \in H(n, m, 3)$

- Choosing up to 3 samples $\Rightarrow$ Full independence among samples

- Choosing loThresh $(>> 3)$ samples $\Rightarrow$ Loss of independence
Why care about Independence

Convergence requires multiplication of probabilities

If every sample is independent => Faster convergence
The principle of principled compromise!

- Choosing up to 3 samples => Full independence among samples

- Choosing loThresh (>> 3) samples => Loss of independence
  - “Almost-Independence” among samples
  - Still provides strong theoretical guarantees of coverage
Strong Guarantees

- \( L = \# \text{ of samples} < |R_F| \)

\[ \frac{L}{(1 + \varepsilon)|R_F|} \leq \Pr[y \text{ is output}] \leq 1.02(1 + \varepsilon) \frac{L}{|R_F|} \]

- **Polynomial** Constant number of SAT calls per sample
  - After one call to ApproxMC
Bug-finding effectiveness

drug frequency $f = 1/10^4$
find bug with probability $\geq 1/2$

<table>
<thead>
<tr>
<th></th>
<th>UniGen</th>
<th>UniGen2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected number of SAT calls</td>
<td>$4.35 \times 10^7$</td>
<td>$3.38 \times 10^6$</td>
</tr>
</tbody>
</table>

An order of magnitude difference!
~20 times faster than UniGen1

![Bar chart comparing Time per sample (s) for UniGen2 and UniGen1 across various benchmarks.](chart.png)

- Benchmarks: s1238a_3_2, s1196a_3_2, s832a_15_7, case_1_b12_2, squaring16, squaring7, doublyLinkedList, LoginService2, Sort, 20.sk, enqueue, Karatsuba, lltraversal, llreverse, diagStencil_new, tutorial3, demo2_new.
### Where are we?

<table>
<thead>
<tr>
<th>Generator</th>
<th>Relative runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-of-the-art: XORSample’</td>
<td>50000</td>
</tr>
<tr>
<td>UniGen</td>
<td>5000</td>
</tr>
<tr>
<td>UniGen1</td>
<td>470</td>
</tr>
<tr>
<td>UniGen2</td>
<td>20</td>
</tr>
<tr>
<td>Ideal Uniform Generator*</td>
<td>10</td>
</tr>
<tr>
<td>SAT Solver</td>
<td>1</td>
</tr>
</tbody>
</table>
The Final Push....

• UniGen requires one time computation of ApproxMC
• Generation of samples in fully distributed fashion (Previous algorithms lacked the above property)
• New paradigms!
Current Paradigm of Simulation-based Verification

- Can not be parallelized since test generators maintain “global state”
- Loses theoretical guarantees (if any) of uniformity
New Paradigm of Simulation-based Verification

- Preprocessing needs to be done only once
- No communication required between different copies of the test generator
- Fully distributed!
## Closing in...

<table>
<thead>
<tr>
<th>Generator</th>
<th>Relative runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State-of-the-art: XORSample’</strong></td>
<td>50000</td>
</tr>
<tr>
<td>UniGen</td>
<td>5000</td>
</tr>
<tr>
<td>UniGen1</td>
<td>470</td>
</tr>
<tr>
<td>UniGen2</td>
<td>20</td>
</tr>
<tr>
<td>Multi-core UniGen2</td>
<td>10 (two cores)</td>
</tr>
<tr>
<td>Ideal Uniform Generator*</td>
<td>10</td>
</tr>
<tr>
<td>SAT Solver</td>
<td>1</td>
</tr>
</tbody>
</table>
So what happened....

Sampling and Counting
Important Applications

Beautiful Theory
But does not work in practice

New Paradigms
(Theory drives practice)

Theoretical Contributions
(Practice drives theory)
Future Directions
Extension to More Expressive domains

• Efficient hashing schemes
  • Extending bit-wise XOR to richer constraint domains provides guarantees but no advantage of SMT progress

• Solvers to handle F + Hash efficiently
  • CryptoMiniSAT has fueled progress for SAT domain
  • Similar solvers for other domains?
Handling Distributions

- Given: CNF formula $F$ and Weight function $W$ over assignments
- Weighted Counting: sum the weight of solutions
- Weighted Sampling: Sample according to weight of solution
- Wide range of applications in Machine Learning
- Extending universal hashing works only in theory so far
Thanks!

Questions?

www.kuldeepmeel.com
kuldeep@rice.edu