Approximate SMT Counting Beyond Discrete Domains

Arijit Shaw
Chennai Mathematical Institute, India
IAI, TCG CREST, Kolkata, India

Kuldeep S. Meel
Georgia Institute of Technology, USA
University of Toronto, Canada

***Abstract*—Satisfiability Modulo Theory (SMT) solvers have advanced automated reasoning, solving complex formulas across discrete and continuous domains. Recent progress in propositional model counting motivates extending SMT capabilities toward model counting, especially for hybrid SMT formulas. Existing approaches, like bit-blasting, are limited to discrete variables, highlighting the challenge of counting solutions projected onto the discrete domain in hybrid formulas.**

We introduce pact, an SMT model counter for hybrid formulas that uses hashing-based approximate model counting to estimate solutions with theoretical guarantees. pact makes a logarithmic number of SMT solver calls relative to the projection variables, leveraging optimized hash functions. pact achieves significant performance improvements over baselines on a large suite of benchmarks. In particular, out of 14,202 instances, pact successfully finished on 603 instances, while Baseline could only finish on 13 instances.

I. Introduction

Propositional model counting is the task of counting the number of satisfying assignments for a given Boolean formula. Recent advances in model counters have made them useful for solving a variety of real-world problems such as probabilistic inference [1], software verification [2], network reliability [3], and neural network verification [4]. The development of model counters has been motivated by the success of SAT solvers over the past few decades, which allowed researchers to explore problems beyond mere satisfiability.

Concurrently, the success of SAT solvers led to an interest in solving the satisfiability of formulas where the variables are not just Boolean. This interest gave rise to the field of Satisfiability Modulo Theories (SMT), which includes a range of theories such as arithmetic, bitvectors, and data structures. SMT theories, inspired by application needs, offer more succinct problem representations than Boolean satisfiability. There has been a significant development in the design of SMT solvers in recent years [5]–[8]. The compactness of SMT and recent solver advancements have made them useful in software and hardware verification [9], [10], security [11], test-case generation, synthesis, planning [12], and optimization [13].

In light of the availability of powerful SMT solvers, a natural next challenge is to explore the problem of model counting for SMT formulas. Despite demonstrated applications [2], [14], SMT counting remains underexplored. Most prior work has concentrated on problems where the underlying theory is discrete, such as bitvectors [15]–[17], linear integers [18]–[20], and strings [21]. Recent studies have shown that in these cases, reducing the problem to Boolean model counting is often the most effective approach [22]. On the other hand, SMT solvers are widely employed for applications that require reasoning over both continuous and discrete variables. For example, to encode hybrid systems in cyber-physical systems or planning, it is essential to have both variables. The existing work in SMT counting is unable to handle SMT formulas with continuous variables.

In this work, we seek to remedy the aforementioned situation. In particular, we focus on a large class of SMT formulas, which we refer to as *hybrid SMT formulas*. The hybrid SMT formulas are defined over both discrete and continuous variables, and we are interested in solutions projected over discrete variables. Our investigations for the development of counting techniques for hybrid SMT formulas are motivated by their ability to model several interesting and relevant applications, such as robustness quantification of cyber-physical systems and counting reachable paths in software (for detailed discussion, see Section I-A).

The primary contribution of this work is the development of a model counting tool, pact[1], for efficient projected counting of hybrid SMT formulas. The framework approximates the model count with (ε, δ) guarantees. pact supports SMT formulas with various theories, including linear and non-linear real numbers, floating-point arithmetic, arrays, bit-vectors, or any combination thereof. The projection variables are over bit-vectors. pact employs a hashing-based approximate model counting technique, utilizing various hash functions such as multiply-mod-prime, multiply-shift, and XOR. The algorithm makes $\mathcal{O}(\log(|S|))$ calls to the SMT oracle, where S is the set of projection variables. We have implemented a user-friendly tool based on CVC5, which will be released post-publication. pact supports a diverse array of theories including QF_ABV, QF_BVFP, QF_UFBV, QF_ABVFPLRA, QF_ABVFP, QF_BVFPLRA.

To demonstrate runtime efficiency, we conduct an extensive empirical evaluation over 14,202 benchmark instances. Out of these 14,202 instances, pact successfully finished on 603 instances, while Baseline could finish only on 13 instances. Importantly, Baseline fails on counts above 3,570, while pact handles instances with over 1.7×10^{19} solutions.

[1]The name pact is an acronym for partition and count for theories.

A. Applications

We now discuss four motivating applications for counting over hybrid SMT formulas.

Robustness Analysis of Automotive Cyber-Physical Systems. Evaluating robustness is crucial in automotive cyber-physical systems (CPS), especially with the rise of autonomous vehicles. Koley et al. [23] encoded the problem using SMT to identify potential CPS attack vectors, incorporating both discrete and continuous variables to represent cybernetic and physical aspects, respectively. This framework extends to a quantitative approach, where the problem becomes an SMT counting query. Robustness is assessed by counting potential attack points, with the projection set defined by the system's input parameters.

Reachability Analysis of Critical Software. Consider a control-flow graph (CFG) of critical software, where we are interested in knowing how many different paths exist in that CFG, such that some violating conditions are reached. We can encode this problem as a counting problem on the SMT formula with discrete and continuous variables, and the projection set would contain Boolean variables indicating whether a node of CFG is reachable. The projected model count will give the number of satisfying paths in CFG.

Quantitative Software Verification. To ensure software reliability, identifying bugs is not always sufficient; a quantitative approach is vital for understanding their impact. A program with an assertion is converted into an SMT formula through a Single Static Assignment (SSA), revealing inputs that lead to assertion failures by counting these specific inputs. Teuber and Weigl [2] reduced the quantitative verification to projected counting over hybrid SMT formulas, wherein the underlying theory is QF_BVFP.

Quantification of Information Flow. In the domain of software reliability, the quantification of information flow represents a critical challenge, particularly in measuring information leakage within industrial software applications. Phan and Malacaria [24] showed that the problem of quantification of information flow in the case of standard programs can be reduced to the task of counting over hybrid SMT formulas defined over QF_BVFP.

II. Preliminaries

Satisfiability Modulo Theory (SMT) [25] combines Boolean satisfiability (SAT) with theories such as integer and real arithmetic, bit-vectors, arrays, enabling efficient and automated analysis of logical formulas involving various data types. *SMT solvers* solves the satisfiability of an SMT formula.

Hybrid SMT Formulas. Some SMT theories are discrete (such as bitvectors and integers), while others are continuous (such as reals and floating points). We define a *hybrid SMT formula* as an SMT formula that combines two or more theories, where there is at least one discrete theory and one continuous theory. For example, a formula in QF_BVLRA is a hybrid formula because it contains both real variables (continuous) and bitvector variables (discrete).

Projection Set and Projected Solutions. Let F represent an SMT formula, where $\mathsf{Vars}(F)$ signifies the set of all variables of F. A *projection set* S is a subset of $\mathsf{Vars}(F)$. Given an assignment τ to $\mathsf{Vars}(F)$, $T_{\downarrow S}$ denotes the projection of τ on S $\mathsf{Sol}(F)$ denotes the set of all solutions to the formula F. $\mathsf{Sol}(F)_{\downarrow S}$ represents the set of all solutions of F projected on S. In the context of this paper, S is a set of discrete variables and therefore, $\mathsf{Sol}(F)_{\downarrow S}$ is a finite set.

Model Counting. Given a formula F and a projection set S, the problem of model counting is to compute $|\mathsf{Sol}(F)_{\downarrow S}|$. An *approximate model counter* takes in a formula F, projection set S, tolerance parameter ε, and confidence parameter δ, and returns c such that $\Pr\left[\frac{|\mathsf{Sol}(F)_{\downarrow S}|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{Sol}(F)_{\downarrow S}|\right] \geq 1-\delta$.

Hash functions. A *hash function* $h : U \to [m]$ maps elements from a universe U to a range $[m] = \{0, 1, \ldots, m-1\}$. A *pairwise independent hash function* is a hash function chosen from a *family* $\mathcal{H}$ of functions $h : U \to [m]$ such that, for any two distinct elements $x_1, x_2 \in U$ and for any $i_1, i_2 \in [m]$: $\Pr[h(x_1) = i_1 \wedge h(x_2) = i_2] = 1/m^2$. A *vector hash function* $h : U^d \to [m]$ extends this concept by mapping d-dimensional vectors of w-bit integers to the range $[m]$. In this paper, the term *hash function* refers to *hash-based constraint*, represented as $h(\mathbf{x}) = \alpha$. The solutions of the formula $F \wedge (h(\mathbf{x}) = \alpha)$ form a subset of the solutions to F, restricted to those where the hash function maps to α.

A. Problem Statement

We shall introduce the projected counting problem, defined by the specific theory of the formula and the projection variables.

Definition 1 ($\mathsf{Count}\mathcal{T}_{\downarrow\mathcal{P}}$(F,S))**.** *Given a logical formula F defined over the SMT theory $\mathcal{T} \cup \mathcal{P}$; and a projection set S on theory $\mathcal{P}$; where $\mathcal{T}$ is either discrete or continuous or combination of both, and $\mathcal{P}$ is a discrete theory,* $\mathsf{Count}\mathcal{T}_{\downarrow\mathcal{P}}(\mathsf{F},\mathsf{S})$ *refers to the problem of counting* $|\mathsf{Sol}(F)_{\downarrow S}|$.

In this work, we consider BV as $\mathcal{P}$. Any possible theory or combination of theories can serve as $\mathcal{T}$. Consequently, the resulting counting problems take forms such as $\mathsf{CountBVFPLRA}_{\downarrow\mathsf{BV}}$, $\mathsf{CountBVFP}_{\downarrow\mathsf{BV}}$, and similar variations.

B. Related Work

The success of propositional model counters, particularly approximate model counters, prompted efforts to extend the techniques to word-level constraints. Chistikov et al. [15] used bit-blasting to extend the propositional model counting technique to word-level benchmarks. Chakraborty et al. [16] designed SMTApproxMC by lifting the hash functions for word-level constraints. Kim and McCamant [17] designed a system to estimate model count of bitvector formulas. Ge et al. [26] developed a probabilistic polynomial-time model counting algorithm for bit-vector problems, and also developed a series of algorithms in the context of related SMT theories to compute or estimate [18], [19], [27] the number of solutions

for linear integer arithmetic constraints. A closely related problem in the hybrid domain of Boolean and rational variables is *Weighted Model Integration* (WMI) [28], which involves computing the volume given the weight density over the entire domain. Extensive research addresses WMI through techniques such as predicate abstraction and All-SMT [29], [30], as well as methods leveraging knowledge compilation [31].

Hashing-based approximate model counting has been extensively studied over the past decades [15], [16], [32]–[41]. Chakraborty et al. [42] showed that variations in a few key components in a generalized framework account for the diversity in prior approaches. While prior works focused on discrete domains such as Boolean variables [35], [36], [40] and bitvectors [15], [16], our approach extends this framework to hybrid SMT formulas.

III. Algorithm and Implementation

We introduce pact, our tool for approximate counting of SMT formulas. It processes a formula F, a set of projection variables S, a *tolerance* ε, and a *confidence* δ to produce an approximation of $|\mathsf{Sol}(F)_{\downarrow S}|$ within the desired tolerance and confidence. The main idea behind pact involves dividing the solution space into equally sized *cells* using hash functions and then enumerating the solutions within each *cell*.

Algorithm 1 pact$(F, S, \varepsilon, \delta, \mathsf{family})$

1: $\mathsf{L} \leftarrow \varnothing, \mathsf{it} \leftarrow 0$
2: $\mathsf{thresh}, \mathsf{numIt}, \ell \leftarrow \mathsf{GetConstants}(\varepsilon, \delta, \mathsf{family})$
3: $\mathsf{C}[0] \leftarrow \mathsf{SaturatingCounter}(F, S, \mathsf{thresh})$
4: **if** $\mathsf{C}[0] \neq \mathsf{T}$ **then return** $\mathsf{C}[0]$
5: **while** $\mathsf{it} < \mathsf{numIt}$ **do**
6: $\quad \mathsf{C} \leftarrow \varnothing, \mathsf{countFound} \leftarrow \bot, i \leftarrow 0$
7: $\quad H \leftarrow \mathsf{GenerateHash}(S, \ell, \mathsf{family})$
8: $\quad$ **while** $\mathsf{countFound} = \bot$ **do**
9: $\quad\quad i \leftarrow \mathsf{NextIndex}(\mathsf{C}, i)$
10: $\quad\quad \mathsf{C}[i] \leftarrow \mathsf{SaturatingCounter}(F \wedge H_{[i]}, P, \mathsf{thresh})$
11: $\quad\quad$ **if** $\mathsf{C}[i] < \mathsf{thresh} \wedge \mathsf{C}[i-1] = \top$ **then**
12: $\quad\quad\quad \mathsf{C}', H' \leftarrow \mathsf{FixLastHash}(F, S, \mathsf{C}, H, i, \ell)$
13: $\quad\quad\quad \mathsf{L}.\mathsf{append}(\mathsf{GetCount}(\mathsf{C}'[i], H'))$
14: $\quad\quad\quad \mathsf{countFound} \leftarrow \top, \mathsf{it}{+}{+}$
15: **return** $\mathsf{FindMedian}(\mathsf{L})$

Algorithm 1 presents the main algorithm pact. The algorithm starts by setting constants of the algorithm, the value for thresh and numIt from the values of ε and δ, depending on the hash family being used. The constants arise from technical calculations in the correctness proof of the algorithm, and the values are shown GetConstants subroutine (Algorithm 3). The value of thresh determines the maximum size of a *cell*. A cell is considered *small* if it has a number of solutions less than this threshold. The value of numIt determines how many times the main loop of the program (lines 5 - 14) is repeated. In each iteration of the main loop, an approximate count is generated, which is stored in the list L. While each of the approximate counts might fail to provide an estimate with the desired δ, the median of the counts of this list gives the approximation of model count with (ε, δ) guarantees.

The main loop of the algorithm begins with the subroutine GenerateHash, which produces a list of hash functions H selected from one of the families $\mathcal{H}_{shift}$, $\mathcal{H}_{prime}$, or $\mathcal{H}_{xor}$ to be used during the current iteration. In each iteration, pact maintains a list C of numbers, where the element $\mathsf{C}[i]$ represents the size of a *cell* after applying the first i hash functions from H, denoted as $H_{[i]}$. The subroutine NextIndex, called in line 9, uses a galloping search to identify an index i in C where the value of $\mathsf{C}[i]$ has been computed. The parameter ℓ in GenerateHash determines the range of hash functions generated. Specifically, (i) for $\mathcal{H}_{shift}$, the range of hash functions is set to 2^ℓ, and (ii) for $\mathcal{H}_{prime}$, GenerateHash constructs hash functions of a range of the smallest prime larger than 2^ℓ.

Following that, in line 10-11, pact checks by a call to SaturatingCounter whether $|\mathsf{Sol}(F \wedge H_{[i]})_{\downarrow S}| < \mathsf{thresh}$ and $|\mathsf{Sol}(F \wedge H_{[i]})_{\downarrow S}| \geq \mathsf{thresh}$. With this condition, pact is enabled a rough estimate of the model count, but to get the estimate within desired error bounds, pact uses the FixLastHash subroutine in line 12. This subroutine eliminates the last hash, $H[i]$, and introduces a new hash function that reduces the number of solution partitions. The subroutine iterates the procedure to find two hash functions h' and h'', such that h'' divides the space into $^k/_2$ parts, while h' divides into k parts. Now if $|\mathsf{Sol}(F \wedge H_{[i-1]} \wedge h')_{\downarrow S}| < thresh, |\mathsf{Sol}(F \wedge H_{[i-1]} \wedge h'')_{\downarrow S}| \geq \mathsf{thresh}$, FixLastHash returns $H' = H_{[i-1]} \wedge h'$ and $C' = |\mathsf{Sol}(F \wedge H_{[i-1]} \wedge h')_{\downarrow S}|$. However, when pact uses $\mathcal{H}_{xor}$, the call to FixLastHash is unnecessary, as it already partitions the solution space into two parts using one hash function.

The subroutine GetCount approximates $\mathsf{Sol}(F)_{\downarrow S}$ by multiplying C' with the number of partitions generated by all the hashes used in H'. This number is then appended to the list L, and pact continues to the next iteration of the main loop. Once the main loop generates count for numIt times, we take the median of all the counts, which is an approximation for $\mathsf{Sol}(F)_{\downarrow S}$ with desired guarantees.

Algorithm 2 FixLastHash$(F, S, \mathsf{C}, H, i, \ell)$

1: **if** family= $\mathcal{H}_{xor}$ **then return** C, H
2: **while** $\ell > 1$ **do**
3: $\quad \ell \leftarrow \lfloor \ell/2 \rfloor$
4: $\quad h^{(\ell)} \leftarrow \mathsf{GenerateHash}(S, \ell, \mathsf{family})$
5: $\quad c \leftarrow \mathsf{SaturatingCounter}(F \wedge H_{[i-1]} \wedge h^{(\ell)}, S, \mathsf{thresh})$
6: $\quad$ **if** $c \neq \top$ **then** $\mathsf{C}[i] = c, H[i] = h^{(\ell)}$
7: $\quad$ **else return** C, H
8: **return** $\bot$

A. Hash functions.

The choice of hash function is one of the most important parts in a hashing-based model counter like pact. In pact, we experiment with three different pairwise independent vector hash functions, which have been used in different literature.

Algorithm 3 GetConstants(ε, δ, family)

1: thresh $\leftarrow 1 + 9.84\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\left(1+\frac{1}{\varepsilon}\right)^2$
2: **if** family $= \mathcal{H}_{xor}$ **then** iters $\leftarrow \lceil 17 \log \frac{3}{\delta} \rceil, \ell \leftarrow 1$
3: **else** iters $\leftarrow \lceil 23 \log \frac{3}{\delta} \rceil, \ell \leftarrow 4$
4: **return** thresh, iters, ℓ

- *Multiply mod prime ($\mathcal{H}_{prime}$)* [43]: For a prime number p, and independent random values $\mathbf{a} = a_0, \ldots, a_{d-1}, b \in [p]$, the hash function from $[p]^d$ to $[p]$ is given by:

$$h_{\mathbf{a},b}(\mathbf{x}) \equiv \left(\sum_{i\in[d]} a_i x_i + b\right) \bmod p = \alpha$$

- *Multiply-shift ($\mathcal{H}_{shift}$)* [44]: For independent random values $\mathbf{a} = a_0, \ldots, a_{d-1}, b \in [2^{\overline{w}}]$, where $\overline{w} \geq w + \ell - 1$ the hash function from $[2^w]^d$ to $[2^\ell]$ is given by:

$$h_{\mathbf{a},b}(\mathbf{x}) \equiv \left(\sum_{i\in[d]} a_i x_i + b\right) [\overline{w} - \ell, \overline{w}) = \alpha$$

- *Bitwise XOR ($\mathcal{H}_{xor}$)* [45]: This is a particular case of the multiply mod prime scheme when $p = 2$.

$$h_{\mathbf{a}}(\mathbf{x}) \equiv \bigoplus_{\{i|a_i=1\}} x_i = \alpha$$

Given the type of hash function to generate, the procedure GenerateHash generates the constraints of the form $h_{\mathbf{a},b} = \alpha$, where α is a randomly chosen element from the domain of the hash function. When this hash function-based constraint H is added to the input formula F, $F \wedge H$ satisfies only those solutions of H for which the hashed value by $H_{\mathbf{a},b}$ is α. therefore, the number of solutions of $F \wedge H$ is approximately p^{th} fraction of number of solution of F. $\mathcal{H}_{prime}$ and $\mathcal{H}_{shift}$ are word-level hash functions that allow us to choose a number of partitions we want to divide our solution space into. When pact uses these hash functions, along with the projection set S, and the family, GenerateHash takes a parameter ℓ, and generates a hash function with domain size p, such that $2^\ell \leq p < 2^{\ell+1}$. Specifically, in $\mathcal{H}_{shift}$, $p = 2^\ell$, in $\mathcal{H}_{prime}$, p is the smallest prime $> 2^\ell$. In case of $\mathcal{H}_{xor}$, $\ell = 1, p = 2$. As S is evident from the context, and $\mathbf{a}, b$ are randomly generated, we use the notation h instead of $h(S)$ to denote the hash functions.

While each of the hashing constraints partitions the solution space into p slices, we often want to partition it into more. To partition the solution space into p^c cells, we use the Cartesian product of c hash functions: $\mathcal{H} \times \mathcal{H} \times \cdots \times \mathcal{H}$. We use the notation $H_{[i]}$ to denote the Cartesian product of the first $i+1$ hash functions, i.e., $H_{[i]} = h_0 \times h_1 \times \cdots \times h_i$.

Slicing. In GenerateHash subroutine of pact, the hash functions have particular domain sizes. But, the bitvectors can have arbitrary width; we *slice* them into bitvectors of smaller width so that the value of (sliced) bitvector lies within the domain of the hash function. Instead of defining hash functions on the variables from S, we define hash functions on *slices* of the variables, defined as follows: For a bitvector $\mathbf{x}$ of width w, we define $\lceil w/\ell \rceil$ slices of width ℓ: $x(\lceil w/\ell \rceil - 1), \ldots, x(1), x(0)$, where $x(i) = x\,[(i+1)\ell - 1 : i\ell]$.

B. Enumerating in a cell.

Another crucial step in pact is to determine when the size of a cell is less than the threshold. pact uses the SaturatingCounter subroutine to enumerate solutions of the formula $F \wedge H_{[i]}$ to determine the size. An SMT solver is employed to find a solution res for the given formula F. The projection of this solution, $\mathsf{res}_{\downarrow S}$, is then *blocked* by adding a constraint $\neg(\mathsf{res}_{\downarrow S})$. Subsequently, the solver is asked to find another solution. This process is repeated in a loop until the solver finds thresh many solutions or reports UNSAT.

C. Searching for the number of partitions.

The number of index of C goes upto $\mathcal{O}(|S|)$. The task for NextIndex subroutine is to find the index, for which the cell size is in the range $[1, \mathsf{thresh}]$. The NextIndex finds the index by $\mathcal{O}(\log(|S|))$ many calls by employing a galloping search method.

We defer the detailed descriptions of GenerateHash, NextIndex, SaturatingCounter, GetCount and to the technical report of the paper.

D. Analysis

Theorem 1. *Let F be a formula defined over a set of variables V and discrete projection variables S. Let* $\mathsf{Est} = \mathsf{pact}(F, S, \varepsilon, \delta)$ *be the approximation returned by* pact*, and* $c = |\mathsf{Sol}(F)_{\downarrow S}|$*. Then,*

$$\Pr\left[\frac{c}{1+\varepsilon} \leq \mathsf{Est} \leq (1+\varepsilon)c\right] \geq 1 - \delta$$

Moreover, pact *makes $\mathcal{O}(\log(|S|)\frac{1}{\varepsilon^2}\log(\frac{1}{\delta}))$ many calls to an SMT solver.*

Proof. We defer the proof to the accompanying technical report. □

E. Impact of Hash Function Families

Our empirical evaluation indicates SaturatingCounter is computationally the most expensive subroutine during execution of pact. Recall that SaturatingCounter is invoked over the formula instance $F \wedge H_{[i]}$; naturally, the choice of the hash function family impacts the practical difficulty of the instance $F \wedge H_{[i]}$. Below, we highlight the tradeoffs offered by different hash function families along many dimensions.

- *Bit-level vs. Bitvector Operations:* The hash function $\mathcal{H}_{xor}$ operates at the bit level, whereas $\mathcal{H}_{shift}$ and $\mathcal{H}_{prime}$ function on bitvectors, making the latter two more amenable to SMT reasoning. This fundamental difference in operation also enables $\mathcal{H}_{prime}$ and $\mathcal{H}_{shift}$ to vary the hash function's domain sizes, a flexibility not available with $\mathcal{H}_{xor}$.
- *Number of hash functions:* As each hash function cannot partition the solution space by more than two partitions, $\mathcal{H}_{xor}$ requires more number of constraints for counting

TABLE I
NUMBER OF INSTANCES COUNTED. (PROJECTION ON BV VARIABLES.)

Logic	Baseline	pact$_{prime}$	pact$_{shift}$	pact$_{xor}$
QF_ABVFPLRA (30)	–	–	–	4
QF_ABVFP (333)	9	1	1	31
QF_ABV (2922)	–	–	–	287
QF_BVFPLRA (55)	–	–	–	37
QF_BVFP (10434)	2	73	88	227
QF_UFBV (428)	2	9	2	17
Total (14202)	13	83	91	603

an instance compared to $\mathcal{H}_{shift}$ and $\mathcal{H}_{prime}$. A number of constraints generally make the problem harder for the solver to solve efficiently.

- *Complexity of required operations:* Multiplication and modulus operations impose significant computational overhead on SMT solvers, making $\mathcal{H}_{prime}$ and $\mathcal{H}_{shift}$ more complicated than $\mathcal{H}_{xor}$. Moreover, when pact uses $\mathcal{H}_{xor}$, it leverages the native XOR reasoning capability of CryptoMiniSat SAT solver inside pact, further increasing the counter's performance.
- *Requirement of bitwidth:* To represent the value of $\sum a_i x_i$, a bitvector of width $2w$ is required in $\mathcal{H}_{shift}$, whereas a bitwidth of $2w + d$ is needed in $\mathcal{H}_{prime}$, as the hash value in $\mathcal{H}_{shift}$ is calculated modulo 2^{2w}, while in $\mathcal{H}_{prime}$ is computed modulo a prime. Since SMT solver performance degrades quickly with increasing bitwidth, $\mathcal{H}_{shift}$ is a more favorable choice in this context. In an alternative implementation of $\mathcal{H}_{prime}$, each term $a_i x_i$ could be represented modulo a prime, but this would require an additional d modulo operations, increasing complexity.

F. Implementation and Supported Theories

We implement pact on top of CVC5, a modern SAT solver. pact uses CVC5 for parsing the formula and running the SMTSolve procedure. In the problem of Count$\mathcal{T}_{\downarrow\mathcal{P}}$, pact solves all of the theories and theory combinations in SMT-Lib for $\mathcal{T}$ and theories of bit-vectors for $\mathcal{P}$. To enhance efficiency, we utilize the SMT solver in its incremental mode, allowing each subsequent query to leverage the information gained from previous calls. Similar incremental calls are different calls to SaturatingCounter at different iterations of pact.

IV. EXPERIMENTAL EVALUATION

We implemented the proposed algorithm on top of state-of-the-art SMT solver CVC5. We used the following experimental setup in the evaluation:

Baseline. As mentioned in Section I, the existing tools for SMT counting are unable to handle Count$\mathcal{T}_{\downarrow\mathcal{P}}$ problems. The situation is similar to the early days of propositional model counting, wherein enumeration-based counters were employed as a baseline. In the same vein, we developed an enumeration-based counter, referred to as Baseline, that uses the state-of-the-art SMT solver, CVC5, the same solver employed by pact. Baseline operates by asking for a solution from the SMT solver and then appending a blocking clause that includes assignments to the projection variable. Subsequently, it asks the solver for another solution, continuing this process until the solver indicates UNSAT.

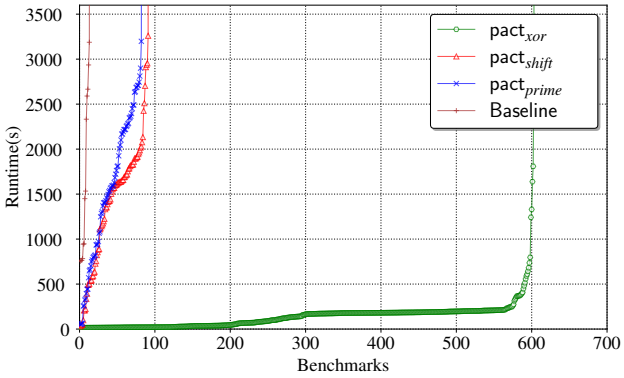

Fig. 1. Comparison of time taken to count by pact and Baseline.

Benchmarks. Our benchmark suite comprises 14,202 instances from the SMT-Lib 2023 release. To minimize bias, we adopted a benchmark selection methodology inspired by early works on propositional model counting. We initially selected all instances supported by six theories. Subsequently, we filtered out instances where the number of solutions was very small (less than 500 models) or where even satisfiability was computationally challenging, as determined by CVC5's inability to find a satisfying assignment within 5 seconds.

Environment. We conducted all our experiments on a high-performance computer cluster, with each node consisting of Intel Xeon Gold 6148 CPUs. We allocated one CPU core and an 8GB memory limit to each solver instance pair. To adhere to the standard timeout used in model counting competitions, we set the timeout for all experiments to 3600 seconds. We use values of $\varepsilon = 0.8$ and $\delta = 0.2$, in line with prior work in the model counting community.

We conduct extensive experiments to understand the following:

RQ1) How does the runtime performance of pact compare to that of Baseline, and how does the performance vary with different hash function families?

RQ2) How accurate is the count computed by pact in comparison to the exact count?

Summary of Results. pact solves a significant number of instances from the benchmarks. It solved 603 instances, while the Baseline counted only 13 instances. Among different hash families, pact performs the best while it uses $\mathcal{H}_{xor}$ hash functions. The accuracy of pact is also noteworthy; the average approximation error is 3.3% while using $\mathcal{H}_{xor}$ hashes.

A. Performance of pact

We evaluate the performance of pact based on two metrics: the number of instances solved and the time taken to solve those instances. To differentiate pact utilizing different hash function families, we use the notations pact$_{prime}$, pact$_{shift}$, and pact$_{xor}$.

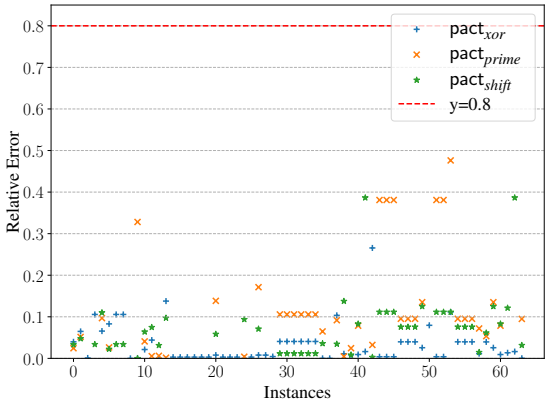

Fig. 2. Accuracy check: observed error in pact vs. the theoretical bound.

Instances solved. For each of the logic, we look at the number of instances solved. Out of 14,202 instances, Baseline could solve only 13 instances. Conversely, pact$_{xor}$ could solve 603 instances, demonstrating a substantial improvement compared to Baseline. The performance varies across different logics, which we represent in Table I. The number of instances solved is 4.2% of the total number of instances, which is expected, given the target problem for these instances is satisfiability, and pact solves a more complex problem.

Comparison of Hash Functions. In the Table I, we compare the performance of pact when it utilizes different hash functions for partitioning the solution space. While all of them perform much better in comparison to Baseline, the best performance is shown by pact$_{xor}$, which solved 603 instances. The performances of pact$_{prime}$ and pact$_{shift}$ are similar, solving 83 and 91 instances.

Solving time comparison. A performance evaluation of Baseline and pact is depicted in Figure 1, which is a cactus plot comparing the solving time. The x-axis represents the number of instances, while the y-axis shows the time taken. A point (i, j) in the plot represents that a solver solved j benchmarks out of the 14,202 benchmarks in the test suite in less than or equal to j seconds. The different curves plot the performance of Baseline and pact with different hash functions.

Performance against the number of solutions. The primary constraint in the enumeration-based method is its capacity for solution counting. The baseline model caps at 3,570 solutions, while pact counts up to 1.7×10^{19}. The performance of pact$_{xor}$ is better than pact$_{shift}$ or pact$_{prime}$ in this regard as well - the maximum count returned by them are in magnitudes of 10^7, while pact$_{xor}$ is of 10^{19}.

B. Quality of Approximation

From our benchmark set, only 13 instances were solved by Baseline. To increase the number of instances for which we know the exact count, we also include the benchmarks with model counts between 100 and 500 in this section of the paper - resulting in 64 instances. We quantify the quality of approximation with the parameter error $e = \max\left(\frac{b}{s}, \frac{s}{b}\right) - 1$, where b is the count from Baseline and s from pact. this definition of error aligns with the ε used in the algorithm's theoretical guarantees and can be interpreted as the observed value of ε. Analysis of all 64 cases found that for pact$_{xor}$, the maximum e to be 0.26 and the average to be 0.03, signifying pact substantially outperforms its theoretical bounds, which is 0.8. In Figure 2, we illustrate the quality of approximation for these instances. The x-axis lists the instances, while the y-axis displays the relative error exhibited by a configuration of pact. A dot (x, y) in the graph indicates x^{th} instance showed a relative error of y. The graph indicates that, for most instances, the error lies below 0.2, with a few instances falling between 0.2 and 0.8. The error for pact$_{shift}$ and pact$_{prime}$ is relatively higher than pact$_{xor}$, with average error being 0.07 and 0.12 and maximum error being 0.39 and 0.48. Our findings underline pact's accuracy and potential as a dependable tool for various applications.

V. Conclusion and Future Work

In this work, we introduced pact, a projected model counter designed for hybrid SMT formulas. Motivated by the diverse applications of model counting and the role of hashing, we explored the impact of various hash functions, examining both bit-level and bitvector-level approaches. Our empirical evaluation demonstrates that pact achieves strong performance on a broad application benchmark set. A dedicated XOR reasoning engine significantly enhanced pact's performance, suggesting that further development of specialized reasoning engines for bit vector-level hash functions could be a promising research direction. Additionally, pact's theoretical framework supports the SMT theory of integers (with specified bounds) as projection variables; this feature is not yet implemented and remains an avenue for future work.

Acknowledgement

We are thankful to Jaroslav Bendik, Supratik Chakraborty, Ashwin Karthikeyan, Aina Niemetz, Mathias Preiner, Uddalok Sarkar, Mate Soos and Jiong Yang for the many useful discussions. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2024-05956]. Part of the work was done when Arijit Shaw was a visiting graduate student at the University of Toronto. Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by Innovation, Science and Economic Development Canada; the Digital Research Alliance of Canada; the Ontario Research Fund: Research Excellence; and the University of Toronto.

References

[1] M. Chavira and A. Darwiche, "On probabilistic inference by weighted model counting," *Artificial Intelligence*, vol. 172, 2008.
[2] S. Teuber and A. Weigl, "Quantifying software reliability via model-counting," in *Proc. of QEST*, 2021.
[3] L. Duenas-Osorio, K. Meel, R. Paredes, and M. Vardi, "Counting-based reliability estimation for power-transmission grids," in *Proc. of AAAI*, 2017.
[4] T. Baluta, S. Shen, S. Shinde, K. S. Meel, and P. Saxena, "Quantitative verification of neural networks and its security applications," in *Proc. of CCS*, 2019.

[5] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. of TACAS*, 2009.

[6] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 SMT solver," in *Proc. of TACAS*, 2013.

[7] H. Barbosa *et al.*, "cvc5: A versatile and industrial-strength smt solver," in *Proc. of TACAS*, 2022.

[8] A. Niemetz and M. Preiner, "Bitwuzla," in *Proc. of CAV*, 2023.

[9] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "Cosa: Integrated verification for agile hardware design," in *Proc. of FMCAD*, 2018.

[10] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Proc. of VSTTE*, 2020.

[11] J. Backes *et al.*, "Stratified abstraction of access control policies," in *Proc. of CAV*, 2020.

[12] M. Cashmore, D. Magazzeni, and P. Zehtabi, "Planning for hybrid systems via satisfiability modulo theories," *Journal of Artificial Intelligence Research*, vol. 67, 2020.

[13] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Communications of the ACM*, vol. 59, no. 2, 2016.

[14] G. Girol, B. Farinier, and S. Bardin, "Not all bugs are created equal, but robust reachability can tell the difference," in *Proc. of CAV*, 2021.

[15] D. Chistikov, R. Dimitrova, and R. Majumdar, "Approximate counting in SMT and value estimation for probabilistic programs," in *Proc. of TACAS*, 2015.

[16] S. Chakraborty, K. Meel, R. Mistry, and M. Vardi, "Approximate probabilistic inference via word-level counting," in *Proc. of AAAI*, 2016.

[17] S. Kim and S. McCamant, "Bit-vector model counting using statistical estimation," in *Proc. of TACAS*, 2018.

[18] C. Ge, F. Ma, X. Ma, F. Zhang, P. Huang, and J. Zhang, "Approximating integer solution counting via space quantification for linear constraints," in *Proc. of IJCAI*, 2019.

[19] C. Ge and A. Biere, "Decomposition strategies to count integer solutions over linear constraints." in *Proc. of IJCAI*, 2021.

[20] C. Ge, "Approximate integer solution counts over linear arithmetic constraints," in *Proc. of AAAI*, 2024.

[21] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *Proc. of CAV*, 2015.

[22] A. Shaw and K. S. Meel, "CSB: A Counting and Sampling Tool for Bitvectors," in *Proc. of SMT Workshop at CAV*, 2024.

[23] I. Koley, S. Dey, D. Mukhopadhyay, S. Singh, L. Lokesh, and S. V. Ghotgalkar, "CAD Support for Security and Robustness Analysis of Safety-critical Automotive Software," *ACM Transactions on Cyber-Physical Systems*, 2023.

[24] Q.-S. Phan and P. Malacaria, "Abstract model counting: a novel approach for quantification of information leaks," in *Proc. of ASIACCS*, 2014.

[25] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2016.

[26] C. Ge, F. Ma, T. Liu, J. Zhang, and X. Ma, "A new probabilistic algorithm for approximate model counting," in *Proc. of IJCAR*, 2018.

[27] C. Ge, F. Ma, and J. Zhang, "VolCE: An Efficient Tool for Solving #SMT(LA) Problems," in *Proc. of PRUV Workshop at IJCAR*, 2018.

[28] V. Belle, A. Passerini, and G. Van den Broeck, "Probabilistic inference in hybrid domains by weighted model integration," in *Proc. of IJCAI*, 2015.

[29] P. Morettin, A. Passerini, and R. Sebastiani, "Efficient weighted model integration via smt-based predicate abstraction," in *Proc. of AAAI*, 2017.

[30] P. Morettin, A. Passerini, and R. Sebastiani, "Advanced smt techniques for weighted model integration," *Artificial Intelligence*, 2019.

[31] S. Kolb, M. Mladenov, S. Sanner, V. Belle, and K. Kersting, "Efficient symbolic integration for probabilistic inference," in *Proc. of IJCAI*, 2018.

[32] D. Achlioptas, Z. Hammoudeh, and P. Theodoropoulos, "Fast sampling of perfectly uniform satisfying assignments," in *Proc. of SAT*, 2018.

[33] D. Achlioptas and P. Theodoropoulos, "Probabilistic model counting with short xors," in *Proc. of SAT*, 2017.

[34] V. Belle, G. Van den Broeck, and A. Passerini, "Hashing-based approximate probabilistic inference in hybrid domains," in *Proc. of UAI*, 2015.

[35] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *Proc. of CP*, 2013.

[36] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls." in *Proc. of IJCAI*, 2016.

[37] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *Proc. of AAAI*, 2006.

[38] M. Soos and K. S. Meel, "BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting," in *Proc. of AAAI*, 2019.

[39] L. Stockmeyer, "The complexity of approximate counting," in *Proc. of STOC*, 1983.

[40] J. Yang and K. S. Meel, "Rounding Meets Approximate Model Counting," in *Proc. of CAV*, 2023.

[41] S. Zhao, S. Chaturapruek, A. Sabharwal, and S. Ermon, "Closing the gap between short and long xors for model counting," in *Proc. of AAAI*, 2016.

[42] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Approximate model counting," in *Handbook of Satisfiability*. IOS Press, 2021.

[43] M. Thorup, "High speed hashing for integers and strings," *arXiv preprint arXiv:1504.06804*, 2015.

[44] M. Dietzfelbinger, "Universal hashing and k-wise independent random variables via integer arithmetic without primes," in *Proc. of STACS*, 1996.

[45] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proc. of STOC*, 1977.