MCBAT: Model Counting for Constraints over Bounded Integer Arrays

Abtin Molavi, Tommy Schneider, Mara Downing, and Lucas Bang(✉)

Harvey Mudd College, Claremont, CA 91711, USA
bang@cs.hmc.edu

Abstract. Model counting procedures for data structures are crucial for advancing the field of automated quantitative program analysis. We present an algorithm and practical tool for performing Model Counting for Bounded Array Theory (MCBAT). As the satisfiability problem for the theory of arrays is undecidable in general, we focus on a fragment of array theory for which we are able to specify an exact model counting algorithm. MCBAT applies to quantified integer array constraints in which all arrays have a finite length. We employ reductions from the theory of arrays to uninterpreted functions and linear integer arithmetic (LIA), and we prove these reductions to be model-count preserving. Once reduced to LIA, we leverage Barvinok's polynomial time integer lattice point enumeration algorithm. Finally, we present experimental validation for the correctness and scalability of our approach and apply MCBAT to a case study on automated average case analysis for array programs, demonstrating applicability to automated quantitative program analysis.

1 Introduction

Model counting is the enabling technology and theory behind automated quantitative program analyses. The ability to count the number of solutions to a constraint allows one to perform reliability analysis [14], probabilistic symbolic execution [17], quantitative information flow analysis [18,23,27,33,35], Bayesian inference [10,11,30], and compiler optimization [29]. Originally stated with respect to Boolean formulas [6], more recent advances in model counting have extended counting capabilities to the theories of linear integer arithmetic [21,34], non-linear numeric constraints [7], strings [2,3,12,22,31,32], word-level counting for bit-vectors applied to the problem of automatic inference [9], and more recent work has begun to combine theories of strings and integers [3]. This paper is the first that we are aware of to directly address model counting for constraints over arrays.

The current space of exploration in model counting is driven by the ubiquity of the types found in common programming languages–Booleans, integers, and strings. In this paper, we expand the space of model countable theories with an algorithm for counting the number of models to constraints over the theory of bounded integer arrays. Model counting for array constraints has practical value in its own right, and also has potential as a basis for future model counting

M. Christakis et al. (Eds.): NSV 2020/VSTTE 2020, LNCS 12549, pp. 1–20, 2020.
https://doi.org/10.1007/978-3-030-63618-0_8

algorithms for other structures that can be modeled as arrays: vectors, maps, hash tables, caches, and so on.

Model counting is a crucial step is quantitative program analysis. For instance, probabilistic symbolic execution (PSE) computes the probability of a program path by counting the number of solutions to the associated path constraints [17]. Quantitative information flow (QIF) analysis often uses PSE to compute probabilistic relationships between program inputs, outputs, observable behaviors, and sensitive information and applies information theoretic metrics to measure the security of an application, design, or protocol [18,23,27,35]. Existing PSE and QIF techniques are either limited to constraints for which there are model counters already available or require ad-hoc model counting approaches [33]. MCBAT provides another tool in the space of model counters which we believe can be useful to quantitative program analysis researchers who encounter integer arrays constraints, as integer arrays are an extremely common data structure. This paper makes the following contributions:

• **Theoretical results.** Our algorithm, MCBAT, composes several model-count preserving reductions to convert an integer array constraint to a formula over linear integer arithmetic (LIA), which can then be model counted using existing algorithms for LIA. We show that our reductions are model-count preserving.

• **Practical tool.** Our tool, also named MCBAT, implements our reductions, applies them to array constraints, and returns the model count.

• **Experimental Validation.** We validated the *correctness* of our implementation by comparing with a baseline alternative implementation that enumerates models using Z3. Both implementations agree on the number of models for all constraints in our benchmark. We evaluated the *scalability* of our algorithm on a benchmark of array formulas with positive results. Finally, we applied our model counting approach in a case study on automatic expected performance analysis.[1]

Example 1. Suppose we want to know the number of sorted integer arrays A of length 3 with array values between 0 and 10. We can express this as an array constraint:

$$\textsc{length}(a) = 3 \wedge \forall i\ 0 \leq a[i] \leq 10 \wedge \forall i\ \forall j\ (i < j\ \rightarrow A[i] \leq A[j])$$

On a technical note, observe that the variable i is universally quantified in the second conjunct. It may be possible that $a[i]$ is an out-of-bounds array index. In our setting, we will take the stance that any indexing that occurs beyond the length of an array results in the same undefined value $\perp$. Now, observe that, ruling out undefined values due to out-of-bounds indexing, the original constraint is equivalent to a set of constraints with three variables a_0, a_1, and a_2 over $\mathbb{Z}$:

$$0 \leq a_0 \leq 10 \wedge 0 \leq a_1 \leq 10 \wedge 0 \leq a_2 \leq 10 \wedge a_0 \leq a_1 \wedge a_1 \leq a_2$$

This constraint, temporarily ignoring the fact that each a_i is an integer, defines a polytope volume P in $\mathbb{R}^3$ (Fig. 1), where each axis corresponds to one of the three a_i(1). The number of solutions to this constraint is then the number of points in $\mathbb{Z}^3 \cap P$. While we do not spell out the details here, one can find that there are 286 integer lattice points in the volume defined by the corresponding polytope. In addition, it is easy to see that the number of models for the original constraint is the same as the number of integer lattice points.

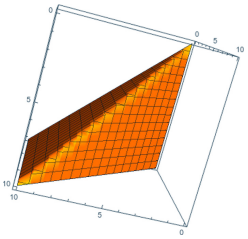

Fig. 1. Polytope defined by constraint of Example 1.

Example 2. Consider a constraint over integer array a and integer variable k.

$$\text{LENGTH}(a) = 2 \wedge (k \geq -15) \wedge \forall i \ (k \leq a[i] \leq 10 \vee k \leq -a[i] \leq 10)$$

This constraint is equivalent to a constraint with three variables $a_0, a_1, k \in \mathbb{Z}$:

$$k \geq -15 \wedge (k \leq a_0 \leq 10 \vee k \leq -a_0 \leq 10) \wedge (k \leq a_1 \leq 10 \vee k \leq -a_1 \leq 10)$$

Similar to our earlier reasoning, this constraint over three variables defines a polytope P in $\mathbb{R}^3$ (Fig. 2). Observe that integer lattice points in P correspond to integer triples (k, a_0, a_1), corresponding to the free variables k and a of the original constraint. Our procedure counts the number of possible models for all free variables in a constraint. For this example, the number of integer lattice points in this polytope, and therefore the number of models to the original constraint, is 10076.

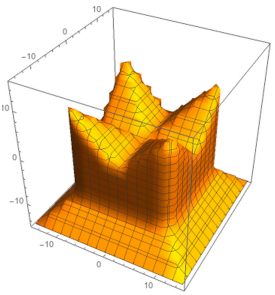

Fig. 2. Polytope defined by constraint of Example 2.

These two examples illustrate the main idea of our approach: count models for an array constraint by transforming it into an instance of lattice point counting within a polytope. While these two examples are easy to visualize, in general, a finite array constraint over integers is model-count equivalent to a set of lattice points in a multi-dimensional polytope.

1.1 Overview

We describe the syntax and semantics of bounded array theory in Sect. 2, which includes quantifiers, Boolean combinations, array terms, integer terms, and linear arithmetic operations and comparisons. In Sect. 3 we describe our model counting algorithm, MCBAT, which relies on a sequence of reductions. These reductions are syntactic transformations applied to a formula f, resulting in a new formula f' that has the same number of solutions as f. These reductions work by returning formulas that (1) contains no array indexing subexpression, (2) contains no array element assignment subexpressions, (3) are quantifier free,

and (4) contains only integer expressions, via Ackermann's reduction [1]. Sects. 3 and 3.2 provide the details of these reductions and proofs that they preserve model counts. The core of our implementation, Sect. 4.2, is written in Haskell and makes calls to the polytope lattice point enumeration library Barvinok. Additionally, Sect. 4.2 describes our experimental benchmark, consisting of array constraints generated from either loop invariant synthesis or symbolic execution. We find that our implementation agrees on the number of models compared to an algorithm which uses Z3 to enumerate models. Finally, we describe a case study in which we perform automatic expected running time analysis on sorting algorithms using MCBAT.

$$
\begin{aligned}
\textit{formula}: \quad & \textit{formula} \wedge \textit{formula} \\
& \mid \textit{formula} \vee \textit{formula} \\
& \mid \textit{formula} \rightarrow \textit{formula} \\
& \mid \neg \textit{formula} \\
& \mid \forall(\textit{int-id}).(\textit{formula}) \\
& \mid \textsc{length}(\textit{array-id}, \mathbb{Z}^{\geq}) \\
& \mid \textit{atom} \\
\textit{atom}: \quad & \textit{term} = \textit{term} \mid \textit{term} < \textit{term} \mid \textit{array} = \textit{array} \\
\textit{array}: \quad & \textit{array-id} \mid \textit{array}\{\textit{term} \leftarrow \textit{term}\} \\
\textit{term}: \quad & \textit{int-id} \mid \mathbb{Z} \mid \mathbb{Z} \times \textit{term} \mid \textit{term} + \textit{term} \mid \textit{array}[\textit{term}]
\end{aligned}
$$

Fig. 3. Abstract grammar for bounded integer array theory constraints.

2 Array Theory: Background, Syntax, and Semantics

The abstract syntax of array constraints that we consider is given in Fig. 3. Our constraint language supports Boolean combinations of formulas, with the expected standard semantics, which may consist of formulas quantified over integer variables, length predicates, or atoms. The formula $\textsc{length}(a, n)$ denotes that the array a has length n. We often write $|a|$ for the length of array a. Atomic expressions may be equality or order comparisons between integer term expressions or equality comparisons between array expressions. Array expressions can be the name of an array (*array-id*) or an array store, written $a\{i \leftarrow e\}$. The notation $a\{i \leftarrow e\}$ represents an array equal to a, except possibly at index i, at which place the array $a\{i \leftarrow e\}$ has the value e. Integer terms can be the names of integer variables (*int-id*), integer constants, products of integer constants and terms, addition of terms, or array index expressions. The term $a[i]$ represents the value stored in the array a indexed at the index i. Note that the arithmetic that makes up the terms is Presburger arithmetic [29]. Note that our grammar here does not enforce that all arrays have a length constraint, but going forward, we will assume that every symbolic array variable a has an associated constraint of the form $\textsc{length}(a, n)$ where n is a positive integer.

Note that while it is *syntactically* possible to access an array outside of its bounds, this is semantically meaningless. As noted in the introduction example, we define indexing that occurs beyond the length of any array to be the undefined value $\perp$.

The semantics of our bounded array theory is that which one would expect from finite length integer arrays found in common programming languages. Our semantics for array store and select expressions departs slightly from the semantics typically employed in the SMT theory of arrays [26]. For instance, the Z3 solver treats arrays as uninterpreted functions from the *entire* set of integers to the array element type: arrays indices in Z3 are unbounded in both the positive and negative directions. Our semantics allows indexes only in the range $[0, \text{LENGTH}(a) - 1]$.

Given a bounded integer array constraint $\phi(a_1, \ldots, a_n, k_1, \ldots, k_w)$, with n symbolic array variables and w integer variables, an interpretation for ϕ is a mapping $I : \{a_1, \ldots, a_n, k_1, \ldots, k_w\} \rightarrow \mathbb{Z}^{|a_1|} \times \ldots \times \mathbb{Z}^{|a_n|} \times \mathbb{Z}^w$ such that when instantiating each free symbol of ϕ with the value defined by I, ϕ evaluates to true. We say that I models ϕ and write $I \models \phi$. The model counting problem to to determine to number of interpretations (models) for a given formula; our goal is to compute $|\{I : I \models \phi\}|$.

3 Model Counting Algorithm: MCBAT

In this section we describe MCBAT, which consists of a series of reductions to linear integer arithmetic, and then show that our reductions are model-count preserving.

3.1 The MCBAT Algorithm

We'll start by presenting an overview of MCBAT (Algorithm 1), and later provide details of important subprocedures (Algorithms 2, 3, 4, and 5).

MCBAT Input. MCBAT takes a formula in the theory of bounded arrays of the form

$$\phi(a_1, \ldots, a_n; k_1, \ldots, k_w) = \text{LENGTH}(a_1, \ell_1) \wedge \cdots \text{LENGTH}(a_n, \ell_n) \wedge \phi_A$$

where ϕ^A is a Boolean combination of quantified array formulas. Here, we've explicitly denoted the n free array-variables and the w free integer-variables. Throughout our algorithm some steps may introduce new free variables, but we ensure that the model count is preserved.

MCBAT Output. We output the number of models there are for $\phi(a_1, \ldots, a_n; k_1, \ldots, k_w)$.

Algorithm 1. MCBAT: Compute the model count for $\phi(a_1, \ldots, a_n; k_1, \ldots, k_w)$

procedure MCBAT($\phi(a_1, \ldots, a_n; k_1, \ldots, k_w)$)
  Decompose ϕ into a tree $\mathbf{T}$ of array formulas $\phi_1, \phi_2, \ldots, \phi_m$.
  Create a tree $\mathbf{T}'$ and a label-formula map $\mathbf{M}$ using $\phi_1, \phi_2, \ldots, \phi_m$ as labels.
  for ϕ_i **do**
    $\phi_i^{(1)} \longleftarrow$ removeTermsInAccess(ϕ_i)
    $\phi_i^{(2)} \longleftarrow$ replaceAllArrayStores($\phi_i^{(1)}$)
    $\phi_i^{(3)} \longleftarrow$ removeQuants($\phi_i^{(2)}$)
  end for
  Construct $\mathbf{M}'$ from $\mathbf{M}$ and the formulas $\phi_1^{(3)}, \ldots, \phi_m^{(3)}$.
  Construct $\phi^{(4)}$ by applying the label-formula map $\mathbf{M}'$ to the Boolean tree $\mathbf{T}'$.
  $\phi^{(5)} \longleftarrow$ AckermannReduction($\phi^{(4)}$)
  return Barvinok($\phi^{(5)}$)
end procedure

High-Level Overview. MCBAT (Algorithm 1) has these main steps:

- Decompose a boolean combination of quantified array formulas into individual quantified array formulas.
- Replace index terms that occur within array access terms with auxiliary integer variables; introduce auxiliary integer constraints to capture this replacement.
- Each array-store term is replaced by equivalent constraints that do not contain array-store expressions.
- Rewrite expressions that are universally quantified over array index variables as a conjunction over all possible indices, with upper bounds enforced by each array s LENGTH predicate.
- Perform Ackermann's reduction, converting array access terms into integer terms.
- Send the resulting linear integer arithmetic constraint to Barvinok to compute the final model count.

Separating Array Sub-formulas. We take a boolean combination of array formulas and decompose it into individual array sub-formulas. We maintain the Boolean skeleton structure of the original input formula, so that after transforming each array sub-formula, we can reconstruct the Boolean combination. Then each of the next steps is performed on individual array sub-formulas. This step is straightforward, we do not provide an algorithm.

Replacing Array Accesses. We replace index terms that occur within array access terms with auxiliary integer variables and introduce auxiliary integer constraints to capture this replacement (Algorithm 2).

Algorithm 2. Replace Array Accesses

```
1: procedure replaceArrayAccesses(φ)
2:     for each unique term t in an array access a[t] do
3:         introduce a new universally quantified variable i_t
4:         conjunct to the antecedent of φ the constraint t = i_t
5:         replace every instance of t with i_t
6:     end for
7: end procedure
```

Replacing Array Stores. Next, each array-store term is replaced by equivalent constraints that do not contain array-store expressions (Algorithm 3).

replaceArrayStores examines a statement in the theory of bounded arrays, and replaces array-stores (i.e., arrays elements of the form $a\{i \leftarrow e\}$) with "fresh" variables. Intuitively, this means that when we encounter an array store operation at index i, we mimic storing a value by using a new array variable to 'imitate' the original array everywhere except for potentially at index i where the new value is e.

Removing Universal Quantifiers. So far, we have a formula containing quantifiers where each array index term is an individual integer variable and there are no array store terms. We then rewrite expressions that are universally quantified over array index variables as a conjunction over all possible indices (Algorithm 4).

Algorithm 3. Replace Array Stores

```
1: procedure replaceArrayStores(φ)
2:     for each array store a{i ← e} do
3:         replace a{i ← e} with the array a', then conjoin the original formula with:
4:         ∀(j).((¬(j = i) → a'[j] = a[j]) ∧ (j = i → a'[j] = e))
5:     end for
6: end procedure
```

Algorithm 4. Remove Quantifiers

```
1: procedure removeQuants(φ)
2:     Let ℓ denote the longest array-length.
3:     while φ is of the form ∀(·).(φ') do
4:         φ ← ⋀_{i∈{0,...,ℓ−1}} φ'
5:     end while
6: end procedure
```

Ackermann's Reduction. Next, we run Ackermann's reduction (Algorithm 5) [1]. Ackermann's reduction is originally phrased as a transformation on terms in the theory of uninterpreted functions. In our setting, observe that we can think of arrays as functions from $\mathbb{Z}$ to $\mathbb{Z}$ and apply the same technique. The reduction

is intended to be satisfiability preserving, but here we use it as a model-count preserving reduction, and show why it is model-count preserving in the following section.

Algorithm 5. Ackermann's Reduction.

1: Input: An array theory formula ϕ^A with array access terms.
2: Output: A linear integer arithmetic formula ϕ^{LIA} that has the same model count.
3: **procedure** AckermannReduction(ϕ^A)
4: Let $\text{FLAT}^{LIA} := \tau(\phi^A)$, where τ replaces the access $a[i]$ with a fresh variable a_i.
5: Let FC^{LIA} denote the following conjunction of *functional consistency* constraints:

$$\text{FC}^{LIA} := \bigwedge_{i \in T} \bigwedge_{j \in T} i = j \rightarrow a_i = a_j$$

where T is the set of all terms used as indices in an array access.
6: **return** ϕ^{LIA} **where** $\phi^{LIA} = \text{FC}^{LIA} \wedge \text{FLAT}^{LIA}$
7: **end procedure**

The fundamental insight behind this reduction is that we can replace any array accesses on single integer variables with new integer variables along with additional *functional consistency constraints*. For example, suppose the terms $a[x], a[y]$, and $a[z]$ occur in a constraint. (Recall that by this point, more complex array index expressions have all been replaced with fresh, individual integer variables.) We can then replace $a[x]$ with a new array value variable a_x, $a[y]$ with a_y, and $a[z]$ with a_z. But now, we need to ensure that if any two variables that were used as indices are ever equal, then the corresponding array value variables must agree. Thus, we also introduce constraints of the form $x = y \rightarrow a_x = a_y$. We introduce such functional consistency constraints for possible pairwise combinations of array index variables.

Model Count the LIA Formula. Barvinok performs model counting by representing a linear integer arithmetic constraint ϕ on variables $X = \{x_1, \ldots, x_n\}$ as a set of symbolic polytopes $\mathscr{P} \subseteq \mathbb{R}^n$. Barvinok's polynomial-time algorithm decomposes $\mathscr{P}$ into a set of n-dimensional 'cones' $\mathscr{K}$, one per vertex of $\mathscr{P}$, computing generating functions that enumerate the set $K \cap \mathbb{Z}^n$ for each $K \in \mathscr{K}$, and then composing the generating functions in order to compute $|\mathscr{P} \cap \mathbb{Z}^n|$, i.e. the number of integer lattice points in the interior of $\mathscr{P}$ and therefore the number of models for ϕ. We have elided many details of Barvinok's algorithm, but the interested reader may consult the provided references for a thorough treatment [4,21,34].

3.2 Correctness

Lemma 1. *Removing access terms is model-count preserving.*

Proof Sketch. Assume an array sub-formula of the form

$$\phi = \forall(i_1, \ldots, i_n).(\phi^A),$$

where ϕ^A contains one or more instances of an array accessed by a term that includes a universally quantified variable. We need to ensure that, when performing quantifier elimination, we do not introduce array accesses that are out-of-bounds. It is straightforward to ensure that if a term is a lone universally quantified variable then that array access, $a[i_1]$, will never be indexed out-of-bounds. If the array has length ℓ, then this will occur exactly when $i_1 < 0$ or $\ell - 1 < i_1$. Thus, when we replace the quantifier $\forall i_1$ with a conjunction over all possible values of i_1, i.e., those from 0 to $\ell - 1$, we introduce new universally quantified variables and in ϕ^A replace old access terms with the new universal variables we set the new universal variable to be equal to the old access term. For example, consider ϕ^A that contains the term $a[2 \times i_3 + 1]$. Then we transform ϕ^A into $\phi'^{AP} \equiv \forall(i_1, \ldots, i_n, j).\phi'^A$ where $\phi'^A = \phi^A \wedge j = 2 \times i_3 + 1$ and each occurrence of $2 \times i_3 + 1$ is replaced by j in ϕ^A. How does this transformation affect the model count? Clearly ϕ'^A is true if and only if $j = 2 \times i_3 + 1$ so the model count is preserved. This holds in general as well, since equating two terms implies that one can substitute them for each other, or in other words, the relevant models are in correspondence.

Lemma 2. *Replacing array stores is model-count preserving.*

Proof Sketch. Next, we remove all the array stores. To do this, we replace the array stores with fresh array variables, and introduce new store-free array formulas to ensure that the fresh array variables operate as the old array stores did. If an array formula contains the array store $a\{i \leftarrow e\}$, then we replace every instance of $a\{i \leftarrow e\}$ with a fresh array variable b. Then, we append to the array formula two more constraints: $b[i] = e$ and $\forall(j).(i = j \rightarrow a[j] = b[j])$.

How does this transformation affect the model count? We're introducing a new free-variable—b—which has the potential to increase the model count. However, the number of models in our new formula will be the same as our old formula because the values of b are completely fixed for any particular values of a, i, and e. Thus, we've ensured our there is only one option for the new variable b for each set of variables that satisfies our original formula.

Lemma 3. *Removing universal quantifiers is model-count preserving.*

Proof Sketch. The next step in our algorithm is to remove the universal quantification from our formula. In particular, we remove quantification of the form $\forall(i).(\phi(i))$, replacing each $\forall$ by conjoining $\phi(i)$ over all values of i such that $\phi(i)$ does not have an array accessed out-of-bounds. If ℓ is the length of an array in $\phi(i)$, then this is $\bigwedge_{i \in \{0,\ldots,\ell-1\}} \phi(i)$.

This rule is carried out exhaustively on each array sub-formula. How does this transformation affect the model count? If a set of variables satisfies our original formula, it clearly satisfies the resulting formula because universal quantification

is the same as conjunction over the integers. Moreover, no statement $\phi(i)$ with $i < 0$ or $i \geq \ell$ has any meaning because $\phi(i)$ will then include an array accessed out-of-bounds. Thus, the model count is preserved.

This step is similar to computing the *index set*, $\mathscr{I}$, of the well-known procedure for SAT-checking array constraints [8]. That computation of the index set computes the minimal set of index variables $\mathscr{I}$ that might be used within array indexing expressions so that satisfiability is maintained when replacing universal quantification with a conjunction over all variables in $\mathscr{I}$. This does not work for model counting, as we need to keep models in correspondence after each transformation.

Lemma 4. *ϕ^A and ϕ^{LIA} as defined in Ackermann's reduction are equisatisfiable.*

Proof Sketch. The equisatisfiable nature of Ackermann's reduction is well-known [1]. We provide a proof sketch giving us the machinery to also claim that Ackermann's reduction is model-count preserving in the next lemma.

Let I^A be a satisfying interpretation for ϕ^A (the input to Algorithm 5). We define I^{LIA}, a satisfying interpretation of ϕ^{LIA} (Algorithm 5 line 6) in the following way. For all variables v that appear in both ϕ^A and ϕ^{LIA}, let $I^{LIA}(v) := I^A(v)$. Then, for the remaining variables in ϕ^{LIA} , let $I^{LIA}(a_i) := I^A(a[i])$. The fact that I^{LIA} is a model for FLAT^{LIA} follows immediately from I^A modeling ϕ^A. Note that FLAT^{LIA} is identical to ϕ^A except that the array access $a[i]$ is replaced with the variable a_i. Since $I^{LIA}(a_i) = I^A(a[i])$, the formulas are identical after substituting variable assignments. Because we may think of array access as a function from $\mathbb{Z}$ to $\mathbb{Z}$, the conjunction of functional consistency constraints FC^{LIA} (Algorithm 5 line 5) is satisfied automatically. Thus their conjunction, ϕ^{LIA}, is satisfied by I^{LIA}.

Now we will assume that I^{LIA} models $\phi^{LIA} = FC^{LIA} \wedge \text{FLAT}^{LIA}$, and show that inverting the process above generates a model for ϕ^A. Let

$$I^A(v) := I^{LIA}(v) \quad \text{and} \quad I^A(a[i]) := I^{LIA}(a_i).$$

We must now argue that I^A models ϕ^A. Note that $\text{FC}^{LIA} \wedge \text{FLAT}^{LIA}$ implies that FC^{LIA} is true and FLAT^{LIA} is true. From the fact that $\text{FLAT}^{LIA} = \tau(\phi^A)$, we have that ϕ^A is true unless I^{LIA} is an assignment such that

$$I^{LIA}(i) = I^{LIA}(j) \wedge \neg(I^{LIA}(a_i) = I^{LIA}(a_j)) \text{ for some } i \text{ and } j.$$

However, I^{LIA} is a model of FC^{LIA} so this cannot be the case. Consequently. Ackermann's reduction preserves satisfiability.

Lemma 5. *Ackerman's reduction is model-count preserving; that is, ϕ^A and ϕ^{LIA} as defined in Ackermann's reduction have the same model count.*

Proof Sketch. Let f be the mapping from a model of ϕ^A to one of ϕ^{LIA} from Lemma 4 and let g be the mapping from a model of ϕ^{LIA} to one of ϕ^A from the

same lemma. Note that $(f \circ g)(I^A) = (g \circ f)(I^A) = I^A$. Therefore, there is a bijection between the set of models of ϕ^A and those of ϕ^{LIA}. The sets have the same cardinality.

Theorem 1. *The MCBAT algorithm is model count preserving.*

Proof Sketch. By Lemmas 1 through 5, the steps to reduce a integer array formula to a formula of linear integer arithmetic are model-count preserving. By appealing to the correctness of the Barvinok algorithm [4], which we then call to produce the final count, MCBAT is a model-count preserving algorithm.

Example 3. Consider a constraint that emerges in work on array-based loop invariant synthesis by Larraz, et al. [20]. Consider a program that, given an length 10 array of integers between −10 and 10 (inclusive), partitions it into two arrays of length 5 where the first contains only nonnegative values and the second contains only negative values. The constraint that emerges from this program is shown here on the left. Applying the steps of MCBAT to this constraint eventually leads to the following mode-count-equivalent LIA formula on the right. Sending this constraint to BARVINOK give a final model count of 62661399052455.

$$
\begin{aligned}
&\text{LENGTH}(b,5) \wedge \\
&\text{LENGTH}(c,5) \wedge \\
&\forall(\alpha).(0 \le \alpha \le i-1 \rightarrow 0 \le b[\alpha]) \wedge \\
&\forall(\alpha).(0 \le \alpha \le j-1 \rightarrow c[\alpha] < 0) \wedge \\
&\forall(k).(-10 \le b[k] \le 10) \wedge \\
&\forall(k).(-10 \le c[k] \le 10) \wedge \\
&0 \le i \le 4 \wedge 0 \le j \le 4
\end{aligned}
\quad \xrightarrow[\text{Reductions}]{\text{MCBAT}} \quad
\begin{aligned}
&\bigwedge_{0\le\alpha\le4} -10 \le b_\alpha \le 10 \ \wedge \\
&\bigwedge_{0\le\alpha\le4} -10 \le c_\alpha \le 10 \ \wedge \\
&\bigwedge_{0\le\alpha\le4} \alpha \le i-1 \rightarrow 0 \le b_\alpha \ \wedge \\
&\bigwedge_{0\le\alpha\le4} \alpha \le j-1 \rightarrow c_\alpha < 0 \ \wedge \\
&0 \le i \le 4 \wedge 0 \le j \le 4
\end{aligned}
$$

4 Experiments and Implementation

4.1 The MCBAT Implementation

We implemented the MCBAT algorithm in a tool, also called MCBAT. A high level architecture of MCBAT can be seen in Fig. 4. The core MCBAT algorithm is implemented in a series of Haskell functions, eventually passing a quantifier-free linear integer arithmetic formula to the Barvinok library which returns the final model count. Array constraints may be entered directly as Haskell expressions and MCBAT also supports reading constraint files in an SMT-LIB2-like format. The complete implementation is freely available along with the source code[1]. In addition, our implementation has an associated Docker image, so that one can immediately download and run MCBAT in a virtual environment using a single terminal command.

[1] Note to reviewers: our implementation and experiments are ready for immediate public release upon publication of our results.

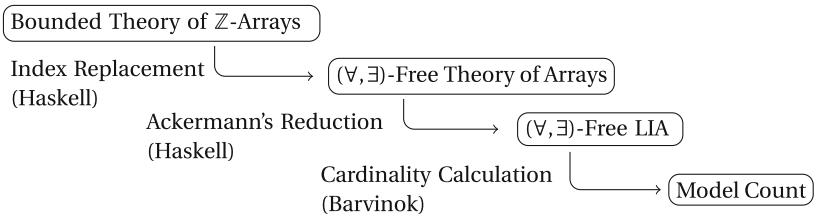

Fig. 4. High-level view of MCBAT implementation architecture.

4.2 MCBAT Experiments

In this section, we give experimental validation of MCBAT's correctness and efficiency and we then describe a case study in which model counting for the theory of arrays is used for automated expected algorithm performance computation. Overall, our experiments have demonstrated that MCBAT is

- **Correct:** MCBAT produces the same model counts compared to a straightforward enumerative approach using Z3, but much faster.
- **Efficiently scalable:** For realistic array sizes and array value domains that may be encountered in practice (up to length 400 with 32-bit array values), MCBAT produces model counts in reasonable amounts of time
- **Applicable:** We applied MCBAT to the problem of automatically computing the average case analysis of well-known array algorithms.

Validating Correctness. We verify that MCBAT computes correct model counts by comparing with a second algorithm using Z3. There is no pre-existing model counting tool for the theory of arrays that we are aware of, so our straightforward baseline comparison is to generate all possible models for an array constraint using Z3 to check satisfiability while incrementing a counter. As mentioned previously in this paper, our array semantics differ slightly from those of Z3: integer arrays in Z3 are a total map from $\mathbb{Z}$ to $\mathbb{Z}$. We accounted for this by automatically constructing additional constraints to require that for any array, $a[i] = 0$ if $i < 0$ or $i > \textsc{length}(a)$. Hence, the models of Z3 and MCBAT are brought into correspondence.

The constraints that we use for this comparison are from the work of Larraz, et al. on automatic generation of loop invariants using SMT for programs operating on arrays [20]. We note that model counting for constraints involving loop invariants is useful in the context of quantitative information flow analysis [18]. Note, we attempted to consistently choose array length, 5, and value range parameters, $[-5, 5]$, so that the Z3 enumerative methods would finish in reasonable time limit of 1 hour. However, three benchmarks (Par, Bin, Copy) did not even finish within one hour using Z3, so we used ranges of $[-1, 1]$ for just those benchmarks in order to complete the correctness comparison. In our experiments, both MCBAT and model enumeration via Z3 computed identical model counts, with MCBAT being significantly faster (Table 1). Consequently, we are confident that MCBAT correctly counts models.

Table 1. Experimental results in which MCBAT and model enumeration using Z3 produce identical counts, along with their running times in seconds.

ID	Description	Length	Range	Count	Z3 time (s)	MCBAT time (s)
HP	Heap property	5	[−5,5]	61226	813	0.02
PI	Partial initialization	5	[−5,5]	1331	327	0.01
Pal	Palindrome	5	[−5,5]	14641	360	0.01
Init	Array initialization	5	[−5, 5]	14641	345	0.01
Ins	Sorted insertion	5	[−5,5]	161051	1097	0.01
Par	Array partition	5	[−1,1]	2916	207	0.02
Bin	Binary search	5	[−1, 1]	1971	140	0.12
Copy	Array copy	5	[−1, 1]	6561	231	0.01
FNN	First not null	5	[−5, 5]	14641	349	0.01
Max	Array maximum	5	[−5,5]	85184	713	<0.01
FO	First occurence	5	[−5,5]	100000	797	0.02
SP	Sum of pairs	5	[−5,5]	83799	1387	0.02
Seq	Seq. initialization	5	[−5,5]	336596	3085	0.01
Shuf	Shuffle	5	[−5,5]	161051	1142	0.27
AC	JutgePaperAC	5	[−5,5]	100000	1002	0.04

Efficiency and Scalability. We applied MCBAT to the same benchmarks that were used to validate the correctness, but with increased array lengths and array value ranges. Considering array lengths of up to 400 and array value ranges as large as $[-2^{32}, 2^{32}-1]$ (i.e. the range of 64-bit signed integers), the enumerative approach using Z3 is not feasible. However, MCBAT is able to compute model counts for all constraints in reasonable amounts of time (Table 2).

Note that MCBAT computes exact model counts, but in Table 2, we report only approximations of model counts in scientific notation up to two significant digits, as the exact model count would not fit in the table (e.g. the count for HP with length 400 and 64-bit integer ranges is almost an 8000 digit number.) With a 10 min timeout limit, we observe the following:

- **Sensitivity to array length.** The dominant bottleneck in our approach is array length. We see that execution time grows significantly as the length increases for a given constraint.
- **Insensitivity to value range.** The running time of MCBAT does not have a high dependence on value range. For example, for the HP benchmark, both value range settings result in extremely similar running times as a function of array length, despite there being large differences in the resulting model counts.

Table 2. McBAT run-time for benchmark constraints. Array lengths run from 10 to 400, and value ranges are $[-2^4,2^4]$ (8-bit integers) and $[-2^{32}, 2^{32}-1]$ (32-bit integers). Times are in seconds, timeout was 10 min. Only the "first" timeout of each experiment set is show, as larger lenghts or ranges also induce a timeout.

ID	Len.	Range	Count	Time
HP	10	$[-2^4,2^4]$	5.33×10^{14}	0.084
HP	50	$[-2^4,2^4]$	2.94×10^{75}	0.413
HP	100	$[-2^4,2^4]$	2.48×10^{151}	2.853
HP	200	$[-2^4,2^4]$	1.76×10^{303}	26.803
HP	400	$[-2^4,2^4]$	8.87×10^{606}	328.46
HP	10	$[-2^{32}, 2^{32}-1]$	1.52×10^{192}	0.101
HP	50	$[-2^{32}, 2^{32}-1]$	6.59×10^{962}	0.392
HP	100	$[-2^{32}, 2^{32}-1]$	1.30×10^{1926}	3.112
HP	200	$[-2^{32}, 2^{32}-1]$	5.09×10^{3863}	31.920
HP	400	$[-2^{32}, 2^{32}-1]$	7.78×10^{7705}	377.50
PI	10	$[-2^4,2^4]$	1.41×10^{12}	0.123
PI	50	$[-2^4,2^4]$	7.74×10^{72}	2.190
PI	100	$[-2^4,2^4]$	6.52×10^{148}	32.659
PI	200	$[-2^4,2^4]$	4.63×10^{300}	598.75
PI	400	$[-2^4,2^4]$	–	t.o.
PI	10	$[-2^{32}, 2^{32}-1]$	1.34×10^{154}	0.141
PI	50	$[-2^{32}, 2^{32}-1]$	5.81×10^{924}	3.934
PI	100	$[-2^{32}, 2^{32}-1]$	1.15×10^{1888}	32.982
PI	200	$[-2^{32}, 2^{32}-1]$	–	t.o.
Pal	10	$[-2^4,2^4]$	4.64×10^{13}	0.113
Pal	50	$[-2^4,2^4]$	2.55×10^{74}	0.319
Pal	100	$[-2^4,2^4]$	2.15×10^{150}	2.358
Pal	200	$[-2^4,2^4]$	1.53×10^{302}	25.442
Pal	400	$[-2^4,2^4]$	–	t.o.
Pal	10	$[-2^{32}, 2^{32}-1]$	2.47×10^{173}	0.125
Pal	50	$[-2^{32}, 2^{32}-1]$	1.07×10^{944}	0.371
Pal	100	$[-2^{32}, 2^{32}-1]$	2.12×10^{1907}	2.330
Pal	200	$[-2^{32}, 2^{32}-1]$	8.28×10^{3833}	24.889
Pal	400	$[-2^{32}, 2^{32}-1]$	–	t.o.
Init	10	$[-2^4,2^4]$	4.64×10^{13}	9.430
Init	50	$[-2^4,2^4]$	–	t.o.
Init	10	$[-2^{32}, 2^{32}-1]$	1.34×10^{154}	243.46
Ins	10	$[-2^4,2^4]$	1.53×10^{15}	0.292
Ins	25	$[-2^4,2^4]$	1.98×10^{37}	15.507
Ins	10	$[-2^{32}, 2^{32}-1]$	4.56×10^{593}	0.265
Ins	25	$[-2^{32}, 2^{32}-1]$	4.45×10^{481}	15.160
Par	10	$[-2^4,2^4]$	1.46×10^{29}	0.099
Par	50	$[-2^4,2^4]$	4.43×10^{150}	10.106
Par	65	$[-2^4,2^4]$	1.59×10^{196}	332.75
Par	10	$[-2^{32}, 2^{32}-1]$	1.30×10^{384}	0.146
Par	50	$[-2^{32}, 2^{32}-1]$	2.44×10^{1925}	8.280
Par	65	$[-2^{32}, 2^{32}-1]$	2.32×10^{2503}	327.55
Shuf	5	$[-2^4,2^4]$	3.91×10^{7}	34.948
Shuf	5	$[-2^{32}, 2^{32}-1]$	2.14×10^{96}	36.528
Bin	10	$[-2^4,2^4]$	1.01×10^{14}	2.473
Bin	25	$[-2^4,2^4]$	2.45×10^{31}	241.96
Bin	10	$[-2^{32}, 2^{32}-1]$	5.39×10^{193}	2.524
Bin	25	$[-2^{32}, 2^{32}-1]$	3.25×10^{476}	287.01
Copy	5	$[-2^4,2^4]$	4.64×10^{13}	0.241
Copy	7	$[-2^4,2^4]$	5.50×10^{19}	5.688
Copy	10	$[-2^4,2^4]$	2.15×10^{27}	509.55
Copy	5	$[-2^{32}, 2^{32}-1]$	2.47×10^{173}	0.366
Copy	7	$[-2^{32}, 2^{32}-1]$	2.86×10^{250}	6.884
FNN	5	$[-2^4,2^4]$	1.19×10^{6}	0.129
FNN	7	$[-2^4,2^4]$	1.29×10^{9}	2.621
FNN	5	$[-2^{32}, 2^{32}-1]$	1.16×10^{77}	0.121
FNN	7	$[-2^{32}, 2^{32}-1]$	3.94×10^{115}	2.600
Max	10	$[-2^4,2^4]$	5.08×10^{15}	0.137
Max	50	$[-2^4,2^4]$	2.79×10^{75}	0.441
Max	100	$[-2^4,2^4]$	2.35×10^{151}	2.769
Max	200	$[-2^4,2^4]$	1.67×10^{303}	31.198
Max	400	$[-2^4,2^4]$	–	t.o.
Max	10	$[-2^{32}, 2^{32}-1]$	1.14×10^{93}	0.093
Max	50	$[-2^{32}, 2^{32}-1]$	4.94×10^{963}	0.382
Max	100	$[-2^{32}, 2^{32}-1]$	9.77×10^{1925}	2.750
Max	200	$[-2^{32}, 2^{32}-1]$	3.82×10^{3852}	31.263
Max	400	$[-2^{32}, 2^{32}-1]$	–	t.o.
FO	10	$[-2^4,2^4]$	1.02×10^{6}	0.329
FO	25	$[-2^4,2^4]$	1.57×10^{28}	18.845
FO	10	$[-2^{32}, 2^{32}-1]$	4.46×10^{189}	0.333
FO	25	$[-2^{32}, 2^{32}-1]$	1.33×10^{474}	16.728
SP	5	$[-2^4,2^4]$	1.20×10^{7}	0.185
SP	7	$[-2^4,2^4]$	3.27×10^{8}	0.467
SP	10	$[-2^4,2^4]$	8.91×10^{11}	2.105
SP	12	$[-2^4,2^4]$	2.89×10^{14}	5.715
SP	5	$[-2^{32}, 2^{32}-1]$	4.27×10^{95}	0.182
SP	7	$[-2^{32}, 2^{32}-1]$	2.92×10^{132}	0.463
SP	10	$[-2^{32}, 2^{32}-1]$	3.62×10^{189}	2.188
SP	12	$[-2^{32}, 2^{32}-1]$	3.92×10^{227}	5.811
Seq	5	$[-2^4,2^4]$	3.91×10^{7}	0.149
Seq	7	$[-2^4,2^4]$	4.26×10^{10}	3.967
Seq	5	$[-2^{32}, 2^{32}-1]$	2.14×10^{97}	0.145
Seq	7	$[-2^{32}, 2^{32}-1]$	7.27×10^{134}	4.007
AC	10	$[-2^4,2^4]$	2.79×10^{12}	2.218
AC	25	$[-2^4,2^4]$	1.06×10^{29}	344.31
AC	10	$[-2^{32}, 2^{32}-1]$	8.91×10^{189}	2.240
AC	25	$[-2^{32}, 2^{32}-1]$	2.65×10^{474}	367.55

Case Study: Average Runtime for Array Algorithms. We now present a case study in which we apply MCBAT to the problem of automatic average case analysis of programs that operate on integer arrays.

Consider a program P that performs operations on integer arrays, an extremely common type of program. For instance, P could be an implementation of a sorting algorithm. Suppose we wish to know what is the average behavior of P over all possible array inputs for a given array length. We can define a cost metric, for example, to be the number of basic code block executed while P runs. We perform symbolic execution on P using a small custom symbolic execution engine for integer array programs written in python. We collect the set of path constraints on symbolic array inputs while tracking the cost of every execution path. Assuming that any array of a given length is equally likely, we compute the relative likelihood of a particular path by performing model counting on the path constraints. Let p_i be the probability of path i and c_i be its cost in terms of the number of executed instructions.

$$E[Cost[P]] = \sum_{i=1}^{n} p_i c_i = \frac{\sum_{i=1}^{n} \mathsf{MCBAT}(\phi_i) \cdot c_i}{\mathsf{MCBAT}\left(\bigvee_{j=1}^{n} \phi_j\right)}$$

For our case study on automatic average case analysis of array algorithms, we implemented the following common procedures and symbolically executed them to produce a set of path constraints Φ and corresponding costs for each program P.

- **Index value check**: Check if an array hold a given value at a given index, $|a| = 10$.
- **Search for constant**: Finds the first index of a value in an array, $|a| = 10$.
- **Search for input parameter**: Finds the first index of an input parameter, $|a| = 10$.
- **Array Comparison**: Lexicographically compares two arrays, $|a| = 200$.
- **Find Max**: Finds the largest value in an array, $|a| = 12$.
- **Sorted Insert**: Performs insert step of insertion sort on a sorted array, $|a| = 14$.
- **Insertion Sort**: Performs an insertion sort on an array, $|a| = 6$.
- **Selection Sort**: Performs a selection sort on an array, $|a| = 6$.
- **Bubble Sort**: Performs a bubble sort on an array, $|a| = 6$.
- **Sorted Merge**: Merges two sorted arrays into a single sorted array, $|a| = 12$.
- **Check Sorted**: Checks whether an array is sorted, $|a| = 50$.

The path constraints were then used to compute $E[\textsc{Cost}(P)]$ using the formula above. The results of this experiment are given in Table 3.

Table 3. Case study results expected computation cost, showing number of path constraints (PC)s the expected cost, symbolic exeuction (SE) time, and time to compute expected cost.

Program	# PCs	E[Cost]	SE time (s)	E[Cost] time (s)
Index Val. Check	2	1	0.0550	12.084
Search for Constant	11	21	57.9044	9.434
Search for Param.	11	21	50.8026	309.530
Array Comparison	301	4.5	107.7987	4401.533
Sorted Insert	14	7.5	0.5920	0.957
Insertion Sort	720	18.5	71.3072	32.288
Select Max	2048	23	252.0237	320.149
Selection Sort	1359	43	653.9933	65.550
Bubble Sort	720	43	$34.464S$	28.385
Sorted Merge	924	22.84	240.8224	110.283
Check Sorted	50	4.44	7.5666	50.813

5 Related Work

The Theory of Arrays. In 1962, McCarthy introduced a formal theory of arrays based the two *select and store* axioms [24]. In more recent times, decision procedures for the theory of arrays have been developed and implemented, for instance in the Z3 SMT solver [25,26]. A comprehensive treatment of satisfiability checking for array constraints is given in Kroening and Strichman's "Decision Procedures: An Algorithmic Point of View" [19].

We find that the most closely related work to ours is that of Plazar, et al [28]. While their work is focused primarily on satisfiability checking of array constraints over arbitrary value types, the bounded fragment of array theory that they focus on and the resulting algorithm bear resemblances to our approaches that focus on bounded integer arrays. While the authors observe a "strong correspondence between the models to the input and transformed formulas", their algorithm, as it is not concerned with model counting, is not strong enough to fully maintain the model correspondence across transformations.

Applications of Array Constraint Procedures. SMT solving for arrays is an important component of symbolic execution for programs that operate on arrays, as in Symbolic Path Finder for Java [16]. Another useful application of satisfiability checking for arrays is the synthesis of invariants over arrays [20] by Larraz, et al., whose constraints we used as a benchmark for our experimental analysis. In this paper, we applied model counting to a case study on computing expected computational cost of functions operating on arrays. Work exists on automatic expected cost computation, based on generating functions rather than model counting algorithms, for algorithms over recursively defined data types [15].

Model Counting. Initial work in model counting applied to formulas of propositional logic. Of particular interest is the use of DPLL as a model counting procedure [6]. Recent years have seen a significant increase in interest in model counting for domains beyond propositional logic. LattE [21] and Barvinok [34] are popular model counters for the theory of linear integer arithmetic. Closely related to the theory of arrays is the theory of strings, which are also an indexable type. Recent approaches to model counting for strings make use of generating functions [22], recurrence relations [32], and automata theory [2]. Earlier work on model counting for data structures exists in which Java code that defines a data structure is symbolically executed and the resulting constraints are model counted using LattE during analysis [13]. Finally, recent theoretical results have been shown for the problem of weighted model counting for constraints containing uninterpreted functions [5].

6 Conclusions and Future Work

We presented our algorithm and practical tool, MCBAT, for performing model counting on constraints over integer arrays of bounded length. MCBAT performs a series of transformations on constraints in order to accomplish model counting, and we showed that these transformations are model-count preserving. In addition, we experimentally validated our approach and demonstrated its usefulness on a case study. It is our hope that MCBAT can be used by other researchers in applications requiring model counting for array constraints.

There are many avenues for future work. Extending our approach to higher dimensional arrays would increase the expressiveness of MCBAT, as would handling arrays of types other than integers. In addition, we would like to allow for reasoning over arrays of symbolic lengths. Finally, as arrays can be used to model vectors, hash maps, memory accesses, heaps, and so on, we model counting for arrays is a first step toward model counting for such more complex data structures.

In performing this research, we observed that in many works on model counting the fundamental insights are to (1) convert elements of satisfiability checking procedure into a model counting procedure or (2) convert elements a model enumeration procedure into a model counting procedure. This is the case with model counting algorithms that are based on DPLL, automata, and generating functions. The challenging question arises: to what degree can satisfiability checking and model enumeration algorithms be converted, perhaps automatically, into counting algorithms?

To conclude, we note that Satisfiability Modulo Theories has dramatically increased the ability to perform program analyses. SMT solvers combine decision procedures for Boolean combinations of constraints from various theories. Because there are model counting algorithms for Boolean formulas, linear integer arithmetic, strings, and now integer arrays, we look forward to a future in which Model Counting Modulo Theories combines model counting procedures for various theories to become the fundamental enabling technology behind quantitative program analysis.

References

1. Ackermann, W.: Solvable Cases of the Decision Problem. North-Holland Pub. Co., Amsterdam (1954)
2. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 255–272. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_15
3. Aydin, A., et al.: Parameterized model counting for string and numeric constraints. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, 04–09 November 2018, pp. 400–410 (2018)
4. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Math. Oper. Res. **19**(4), 769–779 (1994)
5. Belle, V.: Weighted model counting with function symbols. In: Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, 11–15 August 2017 (2017)
6. Birnbaum, E., Lozinskii, E.L.: The good old Davis-Putnam procedure helps counting models. J. Artif. Int. Res. **10**(1), 457–477 (1999)
7. Borges, M., Phan, Q.-S., Filieri, A., Păsăreanu, C.S.: Model-counting approaches for nonlinear numerical constraints. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 131–138. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_9
8. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_28
9. Chakraborty, S., Meel, K., Mistry, R., Vardi, M.: Approximate probabilistic inference via word-level counting, November 2015
10. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6), 772–799 (2008)
11. De Salvo Braz, R., O'Reilly, C., Gogate, V., Dechter, R.: Probabilistic inference modulo theories. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, pp. 3591–3599. AAAI Press (2016)
12. Eiers, W., Saha, S., Brennan, T., Bultan, T.: Subformula caching for model counting and quantitative program analysis. In: Proceedings of The 34th IEEE/ACM International Conference on Automated Software Engineering ASE (2019)
13. Filieri, A., Frias, M.F., Păsăreanu, C.S., Visser, W.: Model counting for complex data structures. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 222–241. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_15
14. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder. In: 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 622–631 (2013)
15. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic average-case analysis of algorithm. Theor. Comput. Sci. **79**(1), 37–109 (1991)
16. Fromherz, A., Luckow, K.S., Pasareanu, C.S.: Symbolic arrays in symbolic pathfinder. ACM SIGSOFT Softw. Eng. Notes **41**(6), 1–5 (2016)
17. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pp. 166–176. ACM, New York (2012)

18. Klebanov, V.: Precise quantitative information flow analysis - a symbolic approach. Theor. Comput. Sci. **538**, 124–139 (2014)
19. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View, 1st edn. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-74105-3
20. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 169–188. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_12
21. Loera, J.A.D., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes. J. Symb. Comput. **38**(4), 1273–1302 (2004)
22. Luu, L., Shinde, S., Saxena, P., Demsky, B.: A model counter for constraints over unbounded strings. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 565–576. ACM, New York (2014)
23. Malacaria, P., Khouzani, M.H.R., Pasareanu, C.S., Phan, Q., Luckow, K.S.: Symbolic side-channel analysis for probabilistic programs. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, 9–12 July 2018, pp. 313–327 (2018)
24. McCarthy, J.: Towards a mathematical science of computation. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds.) Information Processing. SCS, vol. 14, pp. 21–28. Springer, Dordrecht (1962). https://doi.org/10.1007/978-94-011-1793-7_2
25. de Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15–18 November 2009, Austin, Texas, USA pp. 45–52 (2009)
27. Phan, Q., Malacaria, P., Pasareanu, C.S., d'Amorim, M.: Quantifying information leaks using reliability analysis. In: 2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, 21–23 July 2014, pp. 105–108 (2014)
28. Plazar, Q., Acher, M., Bardin, S., Gotlieb, A.: Efficient and complete fd-solving for extended array constraints, pp. 1231–1238, August 2017
29. Pugh, W.: Counting solutions to Presburger formulas: how and why. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 121–134. ACM, New York (1994)
30. Sang, T., Bearne, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: Proceedings of the 20th National Conference on Artificial Intelligence, AAAI 2005, vol. 1, pp. 475–481. AAAI Press (2005)
31. Sherman, E., Harris, A.: Accurate string constraints solution counting with weighted automata. In: Proceedings of The 34th IEEE/ACM International Conference on Automated Software Engineering ASE (2019)
32. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Model counting for recursively-defined strings. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 399–418. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_21
33. Tsiskaridze, N., Bang, L., McMahan, J., Bultan, T., Sherwood, T.: Information leakage in arbiter protocols. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 404–421. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_24

34. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica **48**(1), 37–66 (2007)
35. Visser, W., Pasareanu, C.S.: Probabilistic programming for Java using symbolic execution and model counting. In: Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, 26–28 September 2017, pp. 35:1–35:10 (2017)