Parameterized Model Counting for String and Numeric Constraints

Abdulbaki Aydin[1], Lucas Bang[1], William Eiers[1], Tegan Brennan[1], Miroslav Gavrilov[1], Tevfik Bultan[1], and Fang Yu[2]

[1] University of California, Santa Barbara, USA
[2] National Chengchi University, Taipei, Taiwan

Abstract. Recently, symbolic program analysis techniques have been extended to quantitative and probabilistic program analyses using model counting constraint solvers. Given a constraint and a bound, a model counting constraint solver returns the number of solutions for the given constraint within the given bound. In this paper, we present a parameterized model counting constraint solver for string and numeric constraints. Given a constraint over strings and integers, we first generate automata that accept all solutions to the given constraint. We limit the numeric constraints to linear integer arithmetic, and for non-regular string constraints we over-approximate the solution set. Counting the number of accepting paths in the generated automata solves the model counting problem. Our approach is parameterized in the sense that, we do not assume a finite domain size during automata construction, resulting in a potentially infinite set of solutions, and our model counting approach works for arbitrarily large bounds. We demonstrate the effectiveness of our approach on a large set of string and numeric constraints extracted from software applications.

1 Introduction

Quantitative program analysis arises in many contexts such as probabilistic analysis [9, 12], reliability analysis [10] and quantitative information flow [3, 6, 22, 23, 26]. Developing efficient model counting constraint solvers that can handle complex and diverse set of constraints generated during program analyses is one of the key problems in quantitative program analysis. A model counting constraint solver returns the number of solutions for a given constraint within a given bound [2, 5, 20].

In this paper, we present a model counting constraint solver that can handle both numeric and string constraints. Given a constraint, we construct automata that accept the solutions to the constraint. For numeric constraints, we focus on linear integer arithmetic constraints, and the constructed automata accept a binary encoding of the numbers that satisfy the given numeric constraint. Our approach can handle interactions between numeric and string constraints that arise due to operations such as *length*, which returns the numeric value of the length of a string, and which can be used together with a numeric variable

in a constraint. In order to handle relational constraints, we use multi-track automata that accept tuples of values. Since some string constraints can result in non-regular sets, our automata construction approach over-approximates the solution set in such cases. Hence, our model counting constraint solver provides a sound upper-bound for the the number of solutions for a given constraint.

Our automata-based constraint solving approach reduces the model counting problem to path counting. To count the number of values that satisfy the given constraint within a given domain bound on the string and numeric variables, we count the number of accepting paths in the automata within the path length bound that corresponds to the given domain bound. We use techniques from algebraic graph theory to solve the path counting problem for automata.

We implemented the techniques we present in this paper as part of a tool called Automata Based model Counter (ABC) and we experimented on a large set of constraints generated from symbolic execution of Java and JavaScript programs. We compared our approach with two model counting constraint solvers, one for numeric constraints called LattE [5] and another one for string constraints called SMC [20]. For numeric constraints our tool is as precise as LattE and as fast. For string constraints our tool is faster than SMC and as precise for small constraints, and it is slower than SMC but more precise for larger constraints. Finally, our tool can handle constraints that contain combination of string and numeric variables that neither LattE nor SMC can handle.

Related Work: There has been significant amount of work on string constraint solving in recent years [1, 11, 13, 14, 16–18, 25, 28, 33]; however none of these solvers provide model-counting functionality. Due to the importance of model counting in quantitative program analyses, model counting constraints solvers are gaining increasing attention. SMC is the only other model-counting string constraint solver that we are aware of [20]. Our approach to model counting is strictly more precise than SMC. SMC cannot propagate string values across logical connectives which reduces its precision during model counting, whereas we can handle logical connectives without losing precision. We can also handle complex string operations such as *replace* that SMC cannot handle.

LattE [5] is a model counting constraint solver for counting the number of integer solutions to a formula over linear integer arithmetic. LattE uses the polynomial-time Barvinok algorithm [8] for integer lattice point enumeration. LattE is not able to handle string constraints, so ABC is more expressive than LattE.

While linear algebraic methods for counting paths in a graph are well established, this paper is the first to implement those methods for the purpose of parameterized model counting for relational string and integer constraints. There has been earlier work on integer constraint model counting by counting paths in numeric DFA [21], but this earlier approach can only count models when there are finitely many models. An earlier version of ABC was presented in [2] and was integrated with Symbolic PathFinder (SPF) and applied to side-channel analysis in [6]. These earlier results do not handle combined numeric and string constraints and do not count tuples of solutions for relational constraints.

$$
\begin{aligned}
\varphi &\longrightarrow \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid (\varphi) \mid \varphi_{\mathbb{Z}} \mid \varphi_{\mathbb{S}} \mid \top \mid \bot \\
\varphi_{\mathbb{Z}} &\longrightarrow \beta = \beta \mid \beta < \beta \mid \beta > \beta \\
\varphi_{\mathbb{S}} &\longrightarrow \gamma = \gamma \mid \gamma < \gamma \mid \gamma > \gamma \mid \mathrm{match}(\gamma, \rho) \mid \mathrm{contains}(\gamma, \gamma) \mid \mathrm{begins}(\gamma, \gamma) \mid \mathrm{ends}(\gamma, \gamma) \\
\beta &\longrightarrow v_i \mid n \mid \beta + \beta \mid \beta - \beta \mid \beta \times n \mid (\beta) \\
&\mid\ \mathrm{length}(\gamma) \mid \mathrm{toint}(\gamma) \mid \mathrm{indexof}(\gamma, \gamma) \mid \mathrm{lastindexof}(\gamma, \gamma) \\
\gamma &\longrightarrow v_s \mid \rho \mid \gamma.\gamma \mid (\gamma) \mid \mathrm{reverse}(\gamma) \mid \mathrm{tostring}(\beta) \mid \mathrm{charat}(\gamma, \beta) \mid \mathrm{substring}(\gamma, \beta, \beta) \\
&\mid\ \mathrm{replacefirst}(\gamma, \gamma, \gamma) \mid \mathrm{replacelast}(\gamma, \gamma, \gamma) \mid \mathrm{replaceall}(\gamma, \gamma, \gamma) \\
\rho &\longrightarrow \varepsilon \mid s \mid \rho.\rho \mid \rho|\rho \mid \rho^* \mid (\rho)
\end{aligned}
$$

Fig. 1. Constraint language grammar

In this paper, we extend these earlier results on model counting with the following contributions: 1) an extended constraint language that is more expressive than constraint languages supported by earlier model counting constraint solvers (Section 2), 2) handling of relational string constraints using multi-track automata (Section 3), 3) handling of mixed string and integer constraints using multiple automata (Section 3.3), 4) a technique for extracting length constraints from binary automata (Section 3.3), 5) model counting for tuples of variables (Section 4), 6) heuristics for constraint manipulation that significantly improve the precision and the performance (Section 5), and 7) an extensive experimental evaluation (Section 6).

2 Constraint Language

We present a core string constraint language for representing string and numeric constraints that are generated during program analysis. We define our constraint language using the grammar shown in Fig. 1, where φ denotes a formula, β denotes an integer term, γ denotes a string term, ρ denotes a constant regular expression, v_i denotes an integer variable, v_s denotes a string variable, $\top$ and $\bot$ denote constants true and false, and n denotes an integer constant. We assume Σ denotes the set of all characters (i.e., the alphabet for strings), ε denotes the empty string, and $s \in \Sigma^*$ denotes a string value. A character is a string that has length one. The string operations ".", "|", "$*$" correspond to regular expression operations concatenation, alternation, and Kleene closure, respectively. $<$ and $>$ operations on string terms correspond to lexicographical string comparisons.

A mixed term refers to 1) an integer term that contains string terms as parameters or 2) a string term that contains integer terms as parameters. A mixed constraint refers to a formula that contains one or more mixed terms.

Let $|s|$ denote the length of string s; i.e., $\mathrm{length}(s) = |s|$. The semantics of some of the string operations are shown in Table 1, where t, v, u are string values, i, j are integer values and p is a regular expression where $\mathcal{L}(p)$ denotes the set of strings that match p. We designed this core language to be rich enough to capture common constraints from modern languages such as Java and PHP. String

Table 1. Example string operation definitions

$\text{match}(v, p) \Leftrightarrow v \in \mathcal{L}(p)$	$\text{contains}(v, t) \Leftrightarrow \exists s_1, s_2 \in \Sigma^*: v = s_1 t s_2$
$\text{begins}(v, t) \Leftrightarrow \exists s \in \Sigma^*: v = ts$	$\text{ends}(v, t) \Leftrightarrow \exists s \in \Sigma^*: v = st$
$(\text{indexof}(v, t) = 0 \Leftrightarrow t = \varepsilon) \ \wedge \ (\text{indexof}(v, t) = -1 \Leftrightarrow \neg\text{contains}(v, t)) \wedge$ $(\text{indexof}(v, t) = n \Leftrightarrow \exists s_1, s_2 \in \Sigma^*: v = s_1 t s_2 \wedge \neg\text{contains}(s_1, t) \wedge n = \lvert s_1 \rvert)$	
$\text{charat}(v, i) = t \Leftrightarrow \exists s_0, s_1, \ldots, s_n \in \Sigma: v = s_0 s_1 \ldots s_n \wedge 0 \leq i \leq n \wedge t = s_i$	
$\text{substring}(v, i, j) = t \Leftrightarrow \exists s_1, s_2 \in \Sigma^*: v = s_1 t s_2 \wedge \lvert s_1 \rvert = i \wedge 0 \leq i \leq \lvert v \rvert \wedge \lvert t \rvert = j - i$	
$(\text{replacefirst}(v, t, u) = v \Leftrightarrow \forall s_1, s_2 \in \Sigma^*: \neg\text{match}(v, s_1 t s_2)) \wedge$ $(\text{replacefirst}(v, t, u) = s \Leftrightarrow \exists m, s_1, s_2 \in \Sigma^*: v = s_1 m s_2 \wedge \text{match}(m, t) \wedge (s_1 = \varepsilon \vee$ $(\forall s_3 \in \Sigma^*, s_4 \in \Sigma^+: s_1 = s_3 s_4 \wedge \neg\text{match}(s_3, t) \wedge \neg\text{match}(s_4 m, t))) \wedge$ $(s_2 = \varepsilon \vee (\forall s_5 \in \Sigma^+, s_6 \in \Sigma^*: s_2 = s_5 s_6 \wedge \neg\text{match}(m s_5, t))) \wedge s = s_1 u s_2))$	

operations that are not directly available in our constraint language may be implemented using existing ones. For instance, one can define a generic left trim as $\text{ltrim}(v, p) = s \Leftrightarrow ((\neg\text{begins}(v, p) \wedge s = v) \vee (\text{begins}(v, p) \wedge s = \text{replacefirst}(v, p, \varepsilon)))$.

3 Automata-based Constraint Solving

We use automata-based constraint solving, where the truth set of a formula φ is represented as the language of a multi-track deterministic finite automaton (DFA). Given a formula φ, our goal is to construct an automaton A, such that $\mathcal{L}(A) = [\![\varphi]\!]$, i.e., for any $(x_1, \ldots, x_n) \in [\![\varphi]\!]$, there exists $w \in \mathcal{L}(A)$ such that $w_1 = x_1, \ldots, w_n = x_n$ (where w_i denotes the *ith* track of w) and vice versa. Given a formula, such an automaton may not exist since some string constraints can have non-regular truth sets [32]. Our constraint solver is sound in the sense that it provides an over-approximation of the solution set ($[\![\varphi]\!] \subseteq \mathcal{L}(A)$) when the solution set can not be represented precisely.

Let ω be a formula or a term or a multi- track DFA, $\mathcal{V}(\omega)$ denotes the set of free variables in ω. We define the operation π on automata such that given an automaton A and a target variable set V, $\pi(A, V)$ applies the automata projection operation for $\forall v \in \mathcal{V}(A) \setminus V$, and it applies the automata extend operation for $\forall v \in V \setminus \mathcal{V}(A)$ to get a final automaton where $\mathcal{V}(A) = V$. Given an automaton A as a solution to a formula φ where $\mathcal{L}(A) = [\![\varphi]\!]$ and $\mathcal{V}(A) = \{v_1, \ldots, v_n\}$, and $V = \{v_1, \ldots, v_{n-1}\}$, the result of $\pi(A, V)$, where we project onto all variables but v_n, is an automaton such that $\mathcal{L}(\pi(A, V)) = [\![\exists v_n.\varphi]\!]$. Note that projecting away a variable may result in a non deterministic automaton; the implementation of π operation applies a determinization step after each projection to get a deterministic automaton. Similarly, given an automaton A as a solution to a formula φ where $\mathcal{L}(A) = [\![\varphi]\!]$ and $\mathcal{V}(A) = \{v_1, \ldots, v_{n-1}\}$, and $V = \{v_1, \ldots, v_n\}$, the result of $\pi(A, V)$, where we extend the alphabet as $\Sigma^{n-1} \to \Sigma^n$, is an automaton such that $\mathcal{L}(\pi(A, V)) = \{(x_1, \ldots, x_n) \mid (x_1, \ldots, x_{n-1}) \in [\![\varphi]\!]\}$.

The automata constructor function $\mathcal{A}(\varphi)$ (Algorithm 1) returns an automaton such that $[\![\varphi]\!] \subseteq \mathcal{L}(\mathcal{A}(\varphi))$. Given a formula φ, function $\mathcal{A}$ recursively constructs automata for sub-formulae of φ by using a post order traversal of the

Algorithm 1 Automata constructor: $\mathcal{A}(\varphi)$

1: **if** $\varphi \equiv \varphi_1 \wedge \varphi_2$ **then return** $\text{Refine}(\varphi, \pi(\mathcal{A}(\varphi_1), \mathcal{V}(\varphi)) \cap \pi(\mathcal{A}(\varphi_2), \mathcal{V}(\varphi)))$
2: **else if** $\varphi \equiv \varphi_1 \vee \varphi_2$ **then return** $\pi(\mathcal{A}(\varphi_1), \mathcal{V}(\varphi)) \cup \pi(\mathcal{A}(\varphi_2), \mathcal{V}(\varphi))$
3: **else if** $\varphi \equiv \varphi_{\mathbb{S}}$ **or** $\varphi \equiv \neg\varphi_{\mathbb{S}}$ **then return** $\mathcal{A}_{\mathbb{S}}(\varphi, A)$ ▷ $\mathcal{V}(A) = \mathcal{V}(\varphi_{\mathbb{S}}) \wedge \mathcal{L}(A) = \Sigma^*$
4: **else if** $\varphi \equiv \varphi_{\mathbb{Z}}$ **or** $\varphi \equiv \neg\varphi_{\mathbb{Z}}$ **then return** $\mathcal{A}_{\mathbb{Z}}(\varphi, A)$ ▷ $\mathcal{V}(A) = \mathcal{V}(\varphi_{\mathbb{Z}}) \wedge \mathcal{L}(A) = \Sigma^*$
5: **else if** $\varphi \equiv \neg\varphi$ **then return** $\mathcal{A}(\text{ToNegNormForm}(\varphi))$ ▷ push negation inwards
6: **end if**

syntax tree of the formula. $\mathcal{A}$ calls the $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{Z}}$ functions for constructing automata for string and numeric constraints, respectively. $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{Z}}$ take an automaton characterizing the set of values for the variables appearing the in the formula as input. Initially, all variables are unconstrained. The result of intersection operation is refined using the Refine function for two reasons: 1) to increase the precision by propagating the constraints that cannot be solved precisely, and 2) to propagate the effects of the mixed constraints between integer variables and string variables. We discuss algorithms of $\mathcal{A}_{\mathbb{S}}$, $\mathcal{A}_{\mathbb{Z}}$, and Refine functions in the following sections.

3.1 String Constraint Solving

String automata constructor function $\mathcal{A}_{\mathbb{S}}$ (Algorithm 2) constructs an aligned multi-track DFA [29] for string constraints in our language given a string formula $\varphi_{\mathbb{S}}$ and an automaton A as initial values for the free variables in the formula. First, it constructs automata for the terms appearing in the formula using $\mathcal{A}_{\text{term}}$ function. Next, an automaton is constructed for the string operation (sop) based on the operation semantics. Our string constraint language allows complex string formulae for which we may not be able to construct a precise multi-track DFA that captures the exact solution set. In such cases, we over approximate the solution set in order to guarantee soundness. Negations of string constraints are treated as a special operation to avoid under-approximation. $\mathcal{A}_{\overline{\text{sop}}}$ generates an over approximated solution set for the negated string constraints if it cannot be constructed precisely. Table 2 presents example automata constructions for $\mathcal{A}_{\text{sop}}$ and $\mathcal{A}_{\overline{\text{sop}}}$ functions given the automata constructed for the parameters. Au-

Algorithm 2 String automata constructor: $\mathcal{A}_{\mathbb{S}}(\varphi, A)$

1: **if** $\varphi \equiv \gamma_1 \,\text{sop}\, \gamma_2$ **then** ▷ $\text{sop} \in \{=, <, >, \text{match}, \text{contains}, \text{begins}, \text{ends}\}$.
2: $\quad A \leftarrow \mathcal{A}_{\text{sop}}(\mathcal{A}_{\text{term}}(\gamma_1, A), \mathcal{A}_{\text{term}}(\gamma_2, A))$ ▷ e.g., see Table 2.
3: **else if** $\varphi \equiv \neg(\gamma_1 \,\text{sop}\, \gamma_2)$ **then**
4: $\quad A \leftarrow \mathcal{A}_{\overline{\text{sop}}}(\mathcal{A}_{\text{term}}(\gamma_1), \mathcal{A}_{\text{term}}(\gamma_2))$ ▷ e.g., see Table 2.
5: **end if**
6: **if** $\mathcal{V}(A) \neq \mathcal{V}(\varphi)$ **then** ▷ transform automaton to represent variables in $\mathcal{V}(\varphi)$.
7: $\quad A \leftarrow \pi(\mathcal{A}_{\text{term}'}(\pi(A, \{v_{\gamma_1}\}), \gamma_1), \mathcal{V}(\varphi)) \cap \pi(\mathcal{A}_{\text{term}'}(\pi(A, \{v_{\gamma_2}\}), \gamma_2), \mathcal{V}(\varphi))$
8: **end if**
9: **return** A

Table 2. Example automata constructions of $\mathcal{A}_{\text{sop}}$ and $\mathcal{A}_{\overline{\text{sop}}}$

$\mathcal{A}_{=}(\gamma_1, \gamma_2)$: returns A where $\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 = s_2\}$.
$\mathcal{A}_{<}(\gamma_1, \gamma_2)$: returns A where $\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 < s_2\}$.
$\mathcal{A}_{\text{match}}(\gamma, \rho)$: returns A where $\mathcal{L}(A) = \{s \mid s \in \mathcal{L}(A_\gamma) \wedge s \in \mathcal{L}(A_\rho)\}$.
$\mathcal{A}_{\text{begins}}(\gamma_1, \gamma_2)$: returns A where $\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge \exists t \in \Sigma^*\colon s_1 = s_2 t\}$
$\mathcal{A}_{\text{contains}}(\gamma_1, \gamma_2)$: returns A where $\mathcal{L}(A) = \{(s_1, s_2) \mid s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_2) \wedge s_1 \in \Sigma^* \mathcal{L}(A_{\gamma_2}) \Sigma^* \wedge s_2 \in \mathcal{L}(\text{suffixes}(\text{prefixes}(A_1)))\}$.
$\mathcal{A}_{\overline{\text{contains}}}(\gamma_1, \gamma_2)$: returns A where $\mathcal{L}(A) = \{(s_1, s_2) \mid \exists t_1 \in \mathcal{L}(A_{\gamma_1}), t_2 \in \mathcal{L}(A_{\gamma_2})\colon s_1 \in \mathcal{L}(A_{\gamma_1}) \wedge s_2 \in \mathcal{L}(A_{\gamma_2}) \wedge s_1 \notin \Sigma^* t_2 \Sigma^* \wedge s_2 \notin \mathcal{L}(\text{suffixes}(\text{prefixes}(t_1)))\}$.

tomaton construction for $\neg\text{contains}(\gamma_1, \gamma_2)$ uses suffixes and prefixes functions. Given an automaton A (or a string s), suffixes(A) returns an automaton that accepts all suffixes of the all strings in $\mathcal{L}(A)$. Similarly, given an automaton A, prefixes(A) returns an automaton that accepts all prefixes of the all strings in $\mathcal{L}(A)$. $\mathcal{A}_{\text{sop}}$ and $\mathcal{A}_{\overline{\text{sop}}}$ both generate an automaton where one or more tracks correspond to a string expression (represented with an auxiliary variable, e.g., v_γ), but not a free string variable. In that case, we further process the generated automaton using π and $\mathcal{A}_{\text{term}'}$ to make sure that there is a track generated for each free variable in the input formula (line 7 in Algorithm 2).

We now discuss how $\mathcal{A}_{\text{term}}$ (Algorithm 3) generates automata for the terms in a string formula. $\mathcal{A}_{\text{term}}$ recursively constructs an automaton for any sub term in a term (top). Recursion returns when there is a base term (a literal or a variable). A string formula may contain integer terms. We use length automata [2] for the integer terms used in string terms. ToStrEncoding function at line 6 transforms a binary encoded integer automaton to a length automaton. We discuss the details of ToStrEncoding in Section 3.3. Table 3 shows example term automata constructions given the automata constructed for the parameters. Given a subject automaton A_1 and a search automaton A_2, substrmatch(A_1, A_2) returns a set of state pairs (Q_s, Q_e) as the start and end the states of the matching sub strings of accepting strings. These pairs can be found via automata intersection with matching symbols inserted (proposed for replacement in [30]).

So far we described how we construct automata for string operations (sop). However, the tracks of the constructed automata may correspond to string terms.

Algorithm 3 Term automata constructor: $\mathcal{A}_{\text{term}}(\tau, A)$

```
1: if τ ≡ top(γ_1, ..., γ_n, β_{n+1}, ..., β_m) then      ▷ top ∈ {length, ..., replaceall}.
2:     return A_top(A_term(γ_1), ..., A_tern(β_m))                ▷ e.g., see Table 3.
3: else if τ ≡ ρ then return A_top(ρ)          ▷ returns A where L(A) = {s | s ∈ L(ρ)}.
4: else if τ ≡ n then return A_top(n)           ▷ returns A where L(A) = {s | |s| = n}.
5: else if τ ≡ v_s then return π(A, {v_s}
6: else if τ ≡ v_i then return ToStrEncoding(π(A, {v_s})
7: end if
```

Table 3. Example automata constructions of $\mathcal{A}_{\text{top}}$ and $\mathcal{A}_{\text{top}'}$

$\mathcal{A}_{\text{length}}(\gamma)$: returns A where $\mathcal{L}(A) = \{s \mid \exists t \in \mathcal{L}(A_\gamma) : \lvert s\rvert = \lvert t\rvert\}$.
$\mathcal{A}_{\text{charat}}(\gamma, \beta)$: returns A where $\mathcal{L}(A) = \{s \mid \exists t \in \Sigma^*, q_{\text{init}}, q_e, q_n \in Q_{A_\gamma} : q_n = \delta_{A_\gamma}(q_e, s) \wedge \delta^*_{A_\gamma}(q_n, t) \in F_{A_\gamma} \wedge (q_{\text{init}}, q_e) \in \text{substrmatch}(A_\gamma, A_\beta)\}$.
$\mathcal{A}_{\text{indexof}}(\gamma_1, \gamma_2)$: returns A where $\mathcal{L}(A) = \{s \mid \exists t \in \Sigma^*, q_{\text{init}}, q_e \in Q_{A_{\gamma_1}} : \lvert s\rvert = \lvert t\rvert \wedge q_e = \delta^*_{A_{\gamma_1}}(q_{\text{init}}, t) \wedge (q_{\text{init}}, q_e) \in \text{substrmatch}(A_{\gamma_1}, A_{\gamma_2})\}$.
$\mathcal{A}_{\text{substring}}(\gamma, \beta, \beta)$: returns A where $\mathcal{L}(A) = \{s \mid q_{\text{init}}, q_e, q_n \in Q_{A_\gamma} : q_n = \delta^*_{A_\gamma}(q_e, s) \wedge (q_{\text{init}}, q_e) \in \text{substrmatch}(A_\gamma, A_{\beta_1}) \wedge (q_e, q_n) \in \text{substrmatch}(A_\gamma, A_{\beta_2})\}$.
$\mathcal{A}_{\text{charat}'}(A_{\text{charat}}, \gamma, A_\beta)$: returns A_γ where $\mathcal{L}(A_\gamma) = \{s \mid s \in \mathcal{L}(A_\beta)\mathcal{L}(A_{\text{charat}})\Sigma^*\}$.
$\mathcal{A}_{\text{charat}'}(A_{\text{charat}}, \beta, A_\gamma)$: returns A_β where $A_\beta = \text{indexof}(A_\gamma, A_{\text{charat}})$.

For example, $\varphi_{\mathbb{S}} \equiv \text{charat}(v, 1) = \texttt{"a"}$ constructs an equality automaton with 2-tracks where the first track corresponds to charat term and the second track corresponds to the string literal term (`"a"`). $\mathcal{A}_{\text{term}'}$ accepts the automaton generated for a term and the term itself, and returns an automaton where its tracks correspond to variables ($\{v\}$) in the term. Line 2 in Algorithm 4 recursively computes an automaton for the sub terms. In our example, let A_{v_γ} be the automaton generated for the term charat. First, $\mathcal{A}_{\text{top}'}$ generates an automaton for sub term $\gamma_1 \equiv v$ using A_{v_γ} and $\mathcal{A}_{\text{term}}(\beta_1)$ and the semantics of the charat operation (e.g., see Table 3). Next, we intersect the result of the $\mathcal{A}_{\text{top}'}$ with the automaton $\mathcal{A}_{\text{term}}(\gamma_1)$ to restrict it with the set of possible values for the term. Note that, $\mathcal{A}_{\text{term}}$ can return a cached result that is computed in the previous steps in Algorithm 2. Once the automaton is computed, the recursive step is taken and finally the updated automaton (line 5) for the variable is returned (A_1). For the sub term $\beta_1 \equiv 1$, $\mathcal{A}_{\text{term}'}$ generates the automaton A_2 that accepts the language Σ^* and $\mathcal{V}(A_2) = \emptyset$. Finally, line 3 returns an automaton A where $\mathcal{V}(A) = \{v\}$. An automaton for the right hand side of the formula is generated in the same way as shown in Algorithm 2.

3.2 Integer Constraint Solving

Integer automata constructor function $\mathcal{A}_{\mathbb{Z}}$ (Algorithm 5) handles arithmetic formulae that consist of linear equalities, disequalities, and inequalities ($\varphi_{\mathbb{Z}}$). An

Algorithm 4 Variable automata constructor: $\mathcal{A}_{\text{term}'}(\tau, A)$

1: **if** $\tau \equiv \text{top}(\gamma_1, \ldots, \gamma_n, \beta_{n+1}, \ldots, \beta_m)$ **then**
2: $\quad A_1 \leftarrow \mathcal{A}_{\text{term}'}(\gamma_1, \mathcal{A}_{\text{top}'}(A, \gamma_1, \mathcal{A}_{\text{term}}(\gamma_2), \ldots, \mathcal{A}_{\text{term}}(\beta_m)) \cap \mathcal{A}_{\text{term}}(\gamma_1))), \ldots,$
$\quad A_m \leftarrow \mathcal{A}_{\text{term}'}(\beta_m, \mathcal{A}_{\text{top}'}(A, \beta_m, \mathcal{A}_{\text{term}}(\gamma_1), \ldots, \mathcal{A}_{\text{term}}(\beta_{m-1})) \cap \mathcal{A}_{\text{term}}(\beta_m)))$
3: $\quad$ **return** $\pi(A_1, \mathcal{V}(\tau)) \cap \ldots \cap \pi(A_m, \mathcal{V}(\tau))$
4: **else if** $\tau \equiv n$ **or** $\tau \equiv \rho$ **then return** $\mathcal{A}_{\text{top}'}(\tau)$ ▷ returns a DFA A s.t. $\mathcal{L}(A) = \Sigma^*$.
5: **else if** $\tau \equiv v_s$ **then return** A
6: **else if** $\tau \equiv v_i$ **then return** ToBinEncoding(A) ▷ see Section 3.3.
7: **end if**

Algorithm 5 Integer automata constructor: $\mathcal{A}_{\mathbb{Z}}(\varphi, A)$

1: **if** $\varphi \equiv \text{iop}(\beta_1 \ldots, \beta_n$ **then** ▷ where iop $\in \{=, \neq, >, \geq, \leq, <\}$
2: $A_{\text{iop}} \leftarrow \mathcal{A}_{\text{iop}}(\beta_1, \ldots, \beta_n)$ ▷ construction based on the techniques in [7].
3: **end if**
4: **for each** mixed term $\beta \in \varphi$ **do** ▷ iterate over mixed terms
5: $A_{\text{iop}} \leftarrow A_{\text{iop}} \cap \text{ToBinEncoding}(\mathcal{A}_{\text{term}}(\beta, A))$
6: $A_{\text{iop}} \leftarrow \pi(A_{\text{iop}}, \mathcal{V}(\varphi)) \cap \pi(\mathcal{A}_{\text{term}'}(\beta, \text{ToStrEncoding}(\pi(A_{\text{iop}}, \{v_\beta\}))), \mathcal{V}(\varphi))$
7: **end for**
8: **return** A_{iop}

arithmetic formula is first converted into the form $\sum_{i=1}^{n} a_i \cdot x_i + a_0$ iop 0 where a_i denotes integer coefficients and x_i denotes integer variables. Then, the automata construction techniques that rely on a basic binary adder state machine construction is used to construct an automaton for the arithmetic formula [7].

We track relationship between the integer variables and the string variables if the formula is a mixed constraint (line 4). For any mixed term, an auxiliary variable introduced by $\mathcal{A}_{\text{term}}$ is used to track the values of the mixed term corresponding to satisfying assignments of the string variables in the mixed term. Given that, the generated integer automaton first refined further (line 5) and then extended with string variables that appear in mixed terms (line 6).

3.3 Automata Refinement

To improve the precision, Refine function (Algorithm 6) updates the automata iteratively on every intersection by solving the constraints that may cause over-approximations. There are mainly two cases we consider: 1) string formulae that may have over-approximations due to complexity of the operations and/or mixed terms involved in the formulae 2) integer formulae that contains mixed terms. Refine algorithm calls automata constructor functions $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{Z}}$ given the corresponding formulae and the latest automata computed for the formulae. During automata construction ToStrEncoding function is used to convert binary integer automata to length automata to refine string solutions and ToBinEncoding function is used to convert length of string automata to binary integer automata to refine integer solutions. We adopt algorithms proposed in [31] but further propose BinToSemSet function (Algorithm 7) to construct semilinear sets (instead of finding lower and upper bounds [31]) of binary integer automata.

Given an automaton A and a bit-width bound i on recursion (initially $i = 1$), BinToSemSet(A, i) recursively constructs a semilinear set S, s.t., $\mathcal{L}(A) = [\![S]\!]$ if $\mathcal{L}(A)$ is a semilinear set; $\mathcal{L}(A) \subseteq [\![S]\!]$, otherwise. At recursive steps, once a linear set S is found from a given automaton A, we add the set S to the result of the next recursive call where we pass a new automaton $A \setminus \mathcal{A}(S)$ such that $\mathcal{L}(A \setminus \mathcal{A}(S)) = \mathcal{L}(A) \setminus [\![S]\!]$ and the new bit-width bound is reset to 1. In that recursive step, the algorithm tries to find a linear set that forms from a minimal pair of accepting values in $\mathcal{L}(A)$. The procedure conducts an exhaustive search by enumerating all potential pairs in the set of accepting values that have their

Algorithm 6 Refine(φ, A)

1: **for each** sub formula $\varphi_{\mathbb{S}}$ with possible over-approximation $\in \varphi$ **do**
2: $\quad A \leftarrow A \cap \pi(\mathcal{A}_{\mathbb{S}}(\varphi_{\mathbb{S}}, \pi(A, \mathcal{V}(\varphi_{\mathbb{S}}))), \mathcal{V}(\varphi))$
3: **end for**
4: **for each** sub formula $\varphi_{\mathbb{Z}}$ with possible over-approximation $\in \varphi$ **do**
5: $\quad A \leftarrow A \cap \pi(\mathcal{A}_{\mathbb{Z}}(\varphi_{\mathbb{Z}}, \pi(A, \mathcal{V}(\varphi_{\mathbb{Z}}))), \mathcal{V}(\varphi))$
6: **end for**
7: **return** A

Algorithm 7 BinToSemSet(A, i)

1: **if** $\mathcal{L}(A)$ is a finite set **then return** S, where $[\![S]\!] = \{n \mid n \in \mathcal{L}(A)\}$
2: **else if** $i > 2 \times |A|$ **then return** S, where $[\![S]\!] = \{n \mid (n < 2^i \wedge n \in \mathcal{L}(A)) \vee n > 2^i\}$
3: **else**
4: $\quad N \leftarrow \text{GetValues}(A, i)$
5: $\quad$ **while** $N \neq \emptyset$ **do** $a \leftarrow \text{RemoveMin}(N)$, $N' \leftarrow N$
6: $\quad\quad$ **while** $N' \neq \emptyset$ **do** $b \leftarrow \text{RemoveMin}(N')$
7: $\quad\quad\quad$ **construct** S where $[\![S]\!] = \{n \mid a + (b - a) \times k, k \geq 0\}$
8: $\quad\quad\quad$ **return** $S \cup \text{BinToSemSet}(A \setminus \mathcal{A}(S), 1)$ **if** $[\![S]\!] \subseteq \mathcal{L}(A)$
9: $\quad\quad$ **end while**
10: $\quad$ **end while**
11: $\quad$ **return** $\text{BinToSemSet}(A, i + 1)$
12: **end if**

bit-width bounded by i. GetValues(A, i) returns the set $\{n \mid n < 2^i, n \in \mathcal{L}(A)\}$. If we cannot find a linear set given the bit-width bound i, we increase i by one and recurse the procedure. If the recursion returns at line 1, as a final result we return a semilinar set S where $[\![S]\!] = \mathcal{L}(A)$. If the recursion returns because that bit-width bound is greater than a threshold, we return a semilinear set that over approximates $\mathcal{L}(A)$. The threshold is set as $2 \times |A|$ to ensure that before the termination at least two numbers of any linear set in $\mathcal{L}(A)$ have been checked.

4 Model Counting

Here we describe how to perform parameterized model counting by making use of the automata which result from our constraint solving procedure. A *model* for any formula φ is an assignment of values to all variables such that φ evaluates to true. The *model counting problem* is to count the number of models for a formula φ, which we denote $\#\varphi$. A formula can have infinitely many models. However, we can count the number of models within an infinite space of solutions restricted to a finite range for the free variables. Hence, we perform *parameterized model counting* for string and integer constraints, in which $\#\varphi$ is a function over parameters b_S, which bounds the length of string solutions, and $b_{\mathbb{Z}}$, which bounds the bit-length representation of integer solutions. We write $\#\varphi(b_S, b_{\mathbb{Z}})$ for the parameterized model counting function.

The constraint solving procedure produces a final DFA, A, that contains multi-track sub-automata A_S and $A_{\mathbb{Z}}$ which accept an over-approximation of the tuples of string and integer solutions to φ. The separation of string and integer automata may lose some relational information between string and integer variables, but we can multiply the model counts for each automaton in order to give a sound upper bound on the number of models for tuples of integer and string variables. Note that if we are interested in computing only string models, $\#\varphi(b_S, \infty)$, or only integer models, $\#\varphi(\infty, b_{\mathbb{Z}})$, there is no loss of precision in the model counting procedure. Any loss of precision for strings comes from the over-approximations of non-regular constraints in the solving phase, and for pure integer constraints, the model counting procedure is precise.

We rely on the observation that counting the number of strings of length k in a regular language, $\mathcal{L}$, is equivalent to counting the number of accepting paths of length k in the DFA that accepts $\mathcal{L}$. That is, by using a DFA representation, we reduce the parameterized model counting problem to counting the number of paths of a given length in a graph.

Given a string automaton A_S, we denote the number of string tuples of length k accepted by A_S as $\#f_{A_S}(k)$. Computation of $\#f_{A_S}(k)$ can be done by constructing the transfer matrix of the automaton based on its transition relation [24,27]. Let A_S be a DFA with n states. The transfer matrix T of A is a matrix where $T_{i,j}$ is the number of transitions from state i to state j. The number of paths of length k that start in state i and end in state j is given by $(T^k)_{i,j}$. Then the number of strings of length k accepted by A can be computed using matrix multiplication. We compute uT^kv, where u is the row vector such that $u_i = 1$ if and only if i is the start state and 0 otherwise, and v is the column vector where $v_i = 1$ if and only if i is an accepting state and 0 otherwise. Note that for relational string constraints, the transition alphabet is over tuples of characters and the method described here will count the number of tuples of solutions of a given length. Our counting method is parameterized in the following sense: after a constraint is solved, we can count the number of solutions of any desired size k by computing uT^kv without re-solving the constraint.

The method described above computes $\#f_{A_S}(k)$, the number of string solutions of length exactly k. It is also of interest to compute $\#F_{A_S}(k)$, the number of solutions *within* a given bound. This is accomplished easily by using a common "trick" that is often used to simplify graph algorithms. We add an artificial accepting state s_{n+1} to A_S, resulting in a new DFA A'_S, with λ-transitions from each accepting state to s_{n+1}, and a λ-cycle on s_{n+1}. Then one can see that $\#F_{A_S}(k) = \#F_{A'_S}(k+1)$, and so we apply the transfer matrix method on A'_S.

The method for counting strings of a given length allows us to perform model counting for linear constraints as well. However, we must interpret the bound k in a slightly different manner. A solution DFA $A_{\mathbb{Z}}$ for a set of integer tuples encodes the solutions as bit-strings. Thus, paths of length k in an integer automata correspond to bit string of length k. Since we are using a 2's complement representation with leading sign bits, bit strings of exactly length k correspond to integers in the range $[-2^{k-1}, 2^{k-1})$. Thus, the transfer matrix method allows

$$
\begin{array}{cccc}
\varphi \wedge \varphi \rightarrow \varphi & \varphi \vee \varphi \rightarrow \varphi & \varphi \vee \top \rightarrow \top & \\
\varphi \wedge \top \rightarrow \varphi & \varphi \vee \bot \rightarrow \varphi & \varphi \wedge \bot \rightarrow \bot & \\
 & & & |\epsilon| \rightarrow 0 \\
0 \times \beta \rightarrow 0 & \beta - 0 \rightarrow \beta & \beta = \beta \rightarrow \top & i \neq j \rightarrow \top \\
1 \times \beta \rightarrow \beta & -(-\beta) \rightarrow \beta & \beta \neq \beta \rightarrow \bot & |v_{s_1}.v_{s_2}| \rightarrow |v_{s_1}| + |v_{s_2}| \\
\beta + 0 \rightarrow \beta & \neg(\neg\beta) \rightarrow \beta & i = j \rightarrow \bot & \\
 & & & t.\gamma_1 = t.\gamma_2 \rightarrow \gamma_1 = \gamma_2 \\
\gamma.\epsilon \rightarrow \gamma & \gamma = \gamma \rightarrow \top & t.\gamma_1 \neq v.\gamma_2 \rightarrow \top & \gamma_1.t = \gamma_2.t \rightarrow \gamma_1 = \gamma_2 \\
\epsilon.\gamma \rightarrow \gamma & \gamma \neq \gamma \rightarrow \bot & t.\gamma_1 = v.\gamma_2 \rightarrow \bot & \text{ends}(\gamma_2.\gamma_1, \gamma_1) \rightarrow \top \\
t_1.t_2 \rightarrow t_1 t_2 & t = v \rightarrow \bot & \gamma_1.t \neq \gamma_2.v \rightarrow \top & \text{begins}(\gamma_1.\gamma_2, \gamma_1) \rightarrow \top \\
\gamma \in t \rightarrow \gamma = t & t \neq v \rightarrow \top & \gamma_1.t = \gamma_2.v \rightarrow \bot & \text{contains}(\gamma_2.\gamma_1.\gamma_3, \gamma_1) \rightarrow \top
\end{array}
$$

Fig. 2. Term reduction rules

us to perform model counting over integer domains parameterized by intervals of this form by computing $\#f_{A_{\mathbb{Z}}}(k)$. To count models for arbitrary intervals (a, b), we intersect $A_{\mathbb{Z}}$ with the DFA representing $a \leq x_i \leq b$ for any variable x_i, and then count paths in the resulting DFA.

The methods described above allow us to compute $\#F_{A_S}(k)$ and $\#f_{A_{\mathbb{Z}}}(k)$ independently. Now, we can compute $\#\varphi(b_S, b_{\mathbb{Z}}) = \#F_{A_S}(b_S) \cdot \#f_{A_{\mathbb{Z}}}(b_{\mathbb{Z}})$.

The transfer matrix method relies on computing $uT^k v$ and so we seek to implement an efficient method for computing this product. The time and space complexity trade-offs between various methods of computing $uT^k v$ for counting are well-studied [24, 27] and beyond the scope of this paper. However, we note that one may compute T^k using matrix-matrix multiplication with successive squaring, or one may perform left-to-right vector-matrix multiplication. While successive squaring has a better worst-case time complexity bound, we found that due to typically high sparsity of DFA transfer matrices, it is both faster and less memory intensive to use repeated vector-matrix multiplication.

5 Constraint Simplification Heuristics

In this section we present a set of heuristics for improving both the precision and the efficiency of of our constraint solver.

Term Re-Write Rules: All terms are first reduced with respect to a re-write system based on a set of rules (Fig. 2). These rules include both term normalization rules and tautological simplifications of atomic constraints. Here, i, j are distinct integer constants, t, v are distinct string constants and $\gamma_1, \gamma_2, \gamma_3$ are (not necessarily distinct) string terms.

Dependency Analysis: To reduce the amount of work required to solve a constraint, we note that not all variables of a constraint need be counted together. We define the *constraint graph* of a formula φ to be the graph defined on the set of variables of φ where an edge exists between any two variables if they appear in the same clause of φ. This constraint graph can be decomposed into a finite set of connected components. A connected component C is a maximal subgraph such that if $u, v \in C$ then there exists a path between u and v in C.

$$\gamma_1.\text{contains}(\gamma_2) \to |\gamma_1| \geq |\gamma_2| \qquad \neg\gamma.\text{contains}(t) \to \neg\gamma.\text{begins}(t) \qquad \neg\gamma.\text{ends}(t) \to \gamma \neq t$$
$$\gamma_1.\text{begins}(\gamma_2) \to |\gamma_1| \geq |\gamma_2| \qquad \gamma_1.\gamma_2 = \gamma_3.\gamma_4 \to |\gamma_1| + |\gamma_2| = |\gamma_3| + |\gamma_4|$$
$$\gamma_1.\text{ends}(\gamma_2) \to |\gamma_1| \geq |\gamma_2| \qquad \gamma_1.\gamma_2 = \gamma_3 \to |\gamma_1| + |\gamma_2| = |\gamma_3| \wedge \gamma_3.\text{begins}(\gamma_1)$$

Fig. 3. Implication rules

Constraints on any given variable depend only on variables within its connected component. This allows us to decompose a formula based on connected components, solve and count each component individually, and then take the product of the results to obtain accurate counts for tuples of variables. This results in smaller automata and faster computation.

Equivalence Classes: The variables of a formula φ can be partitioned into equivalence classes so that any pair of given variables x, y are in the same equivalence class only if the have the same solution set. In our implementation, we construct these equivalence classes based on equality clauses. Every term, variable or otherwise, begins in its own equivalence class and for every equality clause, the equivalence classes of the left and right sides are merged.

From each equivalence class, we choose a representative. Priority in this choice is given to constant terms then to variables. Each variable in the equivalence class is then replaced by this representative in the formula φ. This optimization can result in the elimination of variables from φ, and hence tracks from its DFA, without any loss of precision in counting.

Implication Rules: As noted previously, our automata construction for some constraints can be imprecise. However, precision can be improved for some of these constraints by augmenting the original formula φ with clauses implied by φ. We present a set of implication rules which define the augmenting clauses added to φ in the presence of certain imprecise constraints in Fig. 3. We only add a clause to φ if we can solve it precisely and if it can potentially improve the precision for another constraint.

6 Implementation and Experiments

We compare ABC with two existing model counters: (1) SMC [20], a string model counter, and (2) LattE [5, 19], a linear integer arithmetic solver with model counting capabilities. All experiments were run on an Intel i5 machine with 2.5GHz X4 processors and 32GB of memory running Ubuntu 14.04. ABC source code is available online [3] along with the experimental data.

ABC-SMC Comparison for String Constraints: We ran ABC on two benchmarks of satisfiable constraints which were generated via symbolic execution of JavaScript and originally solved with the Kaluza string solver [25]. The authors of SMC translated these benchmarks into their input format and separated them into two sets: SMCSmall and SMCBig. We translated from SMC format to ABC input format. The SMCSmall set contains 17554 test constraints

[3] https://github.com/vlab-cs-ucsb/ABC

and SMCBig contains 1342 test constraints. ABC gives an upper bound (u_{abc}) on the model count for all tuples of string variables, while SMC gives both a lower and upper bound (l_{smc}, u_{smc}), but only for a single target variable of a constraint. Thus, to compare with SMC we project our counts to their target variable. ABC completed SMCSmall after 3.40 minutes (0.01 seconds per constraint) and SMCBig after 6.55 hours (17.60 seconds per constraint); SMC took 2.04 hours (0.42 seconds per constraint) for SMCSmall, and 1.52 hours (4.08 seconds per constraint) for SMCBig. We see that ABC is about 36 times faster than SMC on the small benchmark and 4 times slower on the large benchmark. However, ABC has better precision on 78% of the constraints in the large benchmark. We compare the upper bound ABC gives to those given by SMCfor both date sets, shown in table 4. There were 23 small tests and 15 big tests for which ABC gave a count where $u_{\text{abc}} < l_{\text{smc}}$ or $u_{\text{abc}} > u_{\text{smc}}$. We manually confirmed that ABC gives the correct count for those cases, thus these cases correspond to bugs in SMC's implementation.

Table 4. ABC upper bound (u_{abc}) and SMC lower and upper bounds (l_{smc}, u_{smc}) comparison

Benchmark	**#Constraints**	$l_{\text{smc}} < u_{\text{abc}} < u_{\text{smc}}$	$u_{\text{abc}} = u_{\text{smc}}$	**SMC bugs**
SMCSmall	17554	862 (4.9%)	16669 (95%)	23 (0.1%)
SMCBig	1342	1046 (78%)	281 (20.9%)	15 (1.1%)

ABC-LattE Comparison for Numeric Constraints: We compare ABC with LattE in the context of program analysis using the benchmarks (Table 5) from reliability analysis [10] and side-channel analysis [4, 6]. First nine applications, including sorting algorithms, are benchmarks from reliability analysis [10]. We extended the reliability analysis benchmarks by adding Merge sort, Quick sort, and Binary search examples. Password, LawDB, and CRIME are benchmarks from timing/space side-channel analysis [4, 6].

Both analysis techniques require a symbolic execution tool to extract program path constraints, and a model counting tool to enable quantitative analysis on the path constraints. The implementation of both analysis techniques uses SPF for symbolic execution. Using SPF, we collected path constraints from fifteen applications using the frameworks and we counted solutions to path constraints given bit-length bounds 4,8,16, and 32. ABC and LattE return identical counts for all constraints in all cases as both model counters are precise in counting numeric constraints. We focus on the timing comparison between ABC and LattE.

The LattE input format only supports conjunctions of linear equalities and inequalities. In order to handle disequalities ($\neq$) that can arise from path constraints, a preprocessing step is required. LattE integration with SPF uses Omega [15] to convert disequalities into inequalities, which comes with the benefit of constraint simplifications whenever possible. LattE timing measurements includes

Omega simplification time and SPF simplification time. Details of the LattE integration can found in [4, 6, 10].

Table 5. Average time differences in seconds between ABC and LattE. b is bit-length bound for model counting. Positive means ABC is faster.

Application	**#PCs**	$b_1 = 4$	$b_2 = 8$	$b_3 = 16$	$b_4 = 32$	$\{b_1, b_2, b_3, b_4\}$
Alarm	2000	+0.002	+0.003	+0.003	+0.003	+0.039
Booking	2000	+0.003	+0.003	+0.003	+0.003	+0.043
DaisyChain	1434	+0.235	+0.023	+0.023	+0.022	+0.343
FlapController	641	+0.021	+0.021	+0.021	+0.021	+0.114
RobotGame	660	+0.137	+0.130	+0.130	+0.128	+0.560
Bubble sort	720	+0.004	+0.003	+0.001	−0.004	+0.061
Heap sort	1943	+0.005	+0.005	+0.004	+0.001	+0.066
Insertion sort	720	+0.004	+0.003	+0.001	−0.005	+0.061
Selection sort	1359	+0.005	+0.005	+0.003	−0.000	+0.065
Merge sort	720	+0.004	+0.003	+0.001	−0.005	+0.060
Quick sort	1134	+0.005	+0.005	+0.003	−0.001	+0.065
Binary search	21	−0.705	−0.807	−1.010	−1.653	−1.477
Password	11	+0.066	+0.067	+0.066	+0.064	+0.297
LawDB	101	−2.989	−3.027	−3.088	−3.085	−2.550
CRIME	1540	+0.182	+0.249	+0.252	+0.245	+0.972

Table 5 shows that in general ABC performs better than LattE. As bit-length bound increases the timing differences between ABC and LattE decreases in most of the cases and LattE performs better for some of the applications with the larger bounds. As bit-length bound increases, ABC needs to perform more matrix multiplications which takes more time.

Notice that the timing difference between ABC and LattE, when we count for multiple bounds, is largest; in all cases except Binary search and LawDB. ABC first solves a constraints without putting any bounds on it and then uses generated automaton to count given multiple bounds. In contrast, LattE needs to be called multiple times for each bound.

LattE performs better in counting for Binary search and LawDB applications for all bounds. SPF puts additional ordering constraints on input variables which increases the size of the formulae. The multi-track DFA generated by ABC can be exponential in the size of the input constraints which affects both constraint solving and model counting times. Note that even when we do worse than LattE, the timing difference of the multiple bound count is strictly smaller than the largest single bound count timing difference.

ABC Performance on Mixed String and Numeric Constraints: We evaluated ABC performance on mixed constraints that neither SMC nor LattE can handle. We created a benchmark for mixed constraints using SMCSmall benchmark. Out of 17554 test cases in SMCSmall, 6617 contained length constraints on string variables. Length constraint in SMC benchmarks contains only

integer constants. For every such length constraint, we replaced the constant length with a symbolic integer, thus producing mixed constraints. We ran ABC on all 6617 such constraints, and computed a projected count similarly to the method used for the ABC-SMC comparison. ABC completed after 3.50 minutes (0.03 seconds per constraint). In comparison, ABC averaged 0.01 seconds per constraint for the SMCSmall benchmark, of which the mixed constraints were taken from.

Increasing ABC Performance and Precision with Heuristics: The sizes of the multi-track DFA generated by ABC can be exponential in the size of the input constraints. In our experiments we always use the equivalence class generation and dependency heuristics, since without these heuristics ABC runs out of memory for large formulae. In order to evaluate the effectiveness of the implication heuristic, we run different versions of our tool on the SMCBig benchmark: a version with the implications heuristics and a version without. Both implementations used equivalence class generation and dependency analysis. The results given by each version are shown in figure 4. The version with added implications completed the benchmark after 6.60 hours (17.60 seconds per constraint), while the version without implications took 23.50 minutes (1.05 seconds per constraint). Intuitively, adding implications tends to increase precision, often at the expense of longer execution times. The results reinforce this intuition, at least for this particular benchmark.

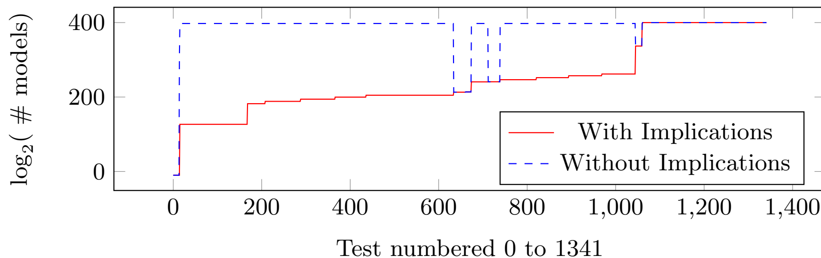

Fig. 4. Precision differences between different configurations for ABC.

7 Conclusion

Model counting is a crucial problem in quantitative program analysis. Using automata as a representation for all solutions of a given constraint reduces the model counting problem to path counting. In this paper, we show that, using automata-based constraint solving, one can construct a model counting constraint solver that is able to handle both string and numeric constraints and their combinations. Our experiments on a large set of constraints generated from Java and JavaScript programs indicate that, automata-based model counting approach is as efficient and as precise as domain specific model counting methods, while it is able to handle a richer set of constraints.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Proceedings of the 26th International Conference on Computer Aided Verification (CAV). pp. 150–166 (2014)
2. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I. pp. 255–272 (2015)
3. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: 30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA. pp. 141–153 (2009)
4. Balasubramanian, D., Luckow, K., Pasareanu, C., Aydin, A., Bang, L., Bultan, T., Gavrilov, M., Kahsai, T., Kersten, R., Kostyuchenko, D., Phan, Q.S., Zhang, Z., Karsai, G.: ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code. In: submission (2017)
5. Baldoni, V., Berline, N., Loera, J.D., Dutra, B., Köppe, M., Moreinis, S., Pinto, G., Vergne, M., Wu, J.: Latte integrale v1.7.2. http://www.math.ucdavis.edu/ latte/
6. Bang, L., Aydin, A., Phan, Q.S., Păsăreanu, C.S., Bultan, T.: String Analysis for Side Channels with Segmented Oracles. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, ACM, New York, NY, USA (2016)
7. Bartzis, C., Bultan, T.: Efficient symbolic representations for arithmetic constraints in verification. Int. J. Found. Comput. Sci. 14(4), 605–624 (2003)
8. 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)
9. Borges, M., Filieri, A., d'Amorim, M., Pasareanu, C.S.: Iterative distribution-aware sampling for probabilistic symbolic execution. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. pp. 866–877 (2015)
10. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in symbolic pathfinder. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 622–631 (2013)
11. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.C.: Word equations with length constraints: What's decidable? In: Proceedings of the 8th International Haifa Verification Conference (HVC). pp. 209–226 (2012)
12. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012. pp. 166–176 (2012)
13. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 188–198 (2009)
14. Hooimeijer, P., Weimer, W.: Solving string constraints lazily. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 377–386 (2010)
15. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega calculator and library, version 1.1. 0. College Park, MD 20742, 18 (1996)
16. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA). pp. 105–116 (2009)

17. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Proceedings of the 9th International Haifa Verification Conference (HVC). pp. 15–31 (2013)
18. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Proceedings of the 26th International Conference on Computer Aided Verification. pp. 646–662 (2014)
19. Loera, J.A.D., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes. Journal of Symbolic Computation 38(4), 1273 – 1302 (2004)
20. Luu, L., Shinde, S., Saxena, P., Demsky, B.: A model counter for constraints over unbounded strings. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). p. 57 (2014)
21. Parker, E., Chatterjee, S.: An automata-theoretic algorithm for counting solutions to presburger formulas. In: Compiler Construction, 13th International Conference, CC 2004, Barcelona, Spain. pp. 104–119 (2004)
22. Phan, Q., Malacaria, P., Pasareanu, C.S., d'Amorim, M.: Quantifying information leaks using reliability analysis. In: Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA. pp. 105–108 (2014)
23. Phan, Q., Malacaria, P., Tkachuk, O., Pasareanu, C.S.: Symbolic quantitative information flow. ACM SIGSOFT Software Engineering Notes 37(6), 1–5 (2012)
24. Ravikumar, B., Eisman, G.: Weak minimization of DFA - an algorithm and applications. Theor. Comput. Sci. 328(1-2), 113–133 (2004)
25. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: Proceedings of the 31st IEEE Symposium on Security and Privacy (2010)
26. Smith, G.: On the foundations of quantitative information flow. In: Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, York, UK, March 22-29, 2009. Proceedings. pp. 288–302 (2009)
27. Stanley, R.P.: Enumerative Combinatorics: Volume 1. Cambridge University Press, New York, NY, USA, 2nd edn. (2011)
28. Trinh, M., Chu, D., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1232–1243 (2014)
29. Yu, F., Alkhalaf, M., Bultan, T.: Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. pp. 605–609. ASE '09, IEEE Computer Society, Washington, DC, USA (2009)
30. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods in System Design 44(1), 44–70 (2014)
31. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: Combining string analysis and size analysis. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,. pp. 322–336. TACAS '09, Springer-Verlag, Berlin, Heidelberg (2009)
32. Yu, F., Bultan, T., Ibarra, O.H.: Relational string verification using multi-track automata. Int. J. Found. Comput. Sci. 22(8), 1909–1924 (2011)
33. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A z3-based string solver for web application analysis. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). pp. 114–124 (2013)