Validity Checking for Finite Automata over Linear Arithmetic Constraints *

Gary Wassermann and Zhendong Su

Department of Computer Science, University of California, Davis
{wassermg,su}@cs.ucdavis.edu

Abstract Decision procedures underlie many program analysis problems. Traditional program analysis algorithms attempt to prove some property about a single, statically-defined program by generating a single constraint. Accordingly, traditional decision procedures take single constraints as input. Extending these traditional program analysis algorithms to reason about potentially infinite languages of programs (as generated by a given metaprogram) requires a new class of decision procedures that reason about languages of constraints. This paper introduces the parameterized class of validity checking problems that take as input a language generator $\mathcal{A}$. The parameters are: (1) the language formalism for $\mathcal{A}$, (2) the theory under which each string in the language of $\mathcal{A}$ is interpretted, and (3) the quantification (existential/universal) of the constraints in the language to which the validity property applies. We introduce such decision problems by presenting an algorithm that decides whether a given finite state automaton $\mathcal{A}$ generates any valid linear arithmetic constraints.

1 Introduction

Many program analysis and formal verification problems reduce to validity or satisfiability checking over some logical theories. Consequently, significant effort has been devoted to designing efficient decision procedures for these theories. Traditional program analysis problems address individual programs, so the decision procedures that underlie program analysis algorithms take a single constraint φ. Extending program analysis problems to address potentially infinite languages of programs (as generated by a metaprogram) requires decision procedures that take languages of constraints. We introduce the study of such decision procedures in this paper. The input to decision procedures over languages of constraints is a language generator $\mathcal{A}$, where each string in the language of $\mathcal{A}$ is a constraint in a given theory. The problem such procedures address is: Does there exist a valid constraint in the language of $\mathcal{A}$, or alternatively, are all constraints in the language of $\mathcal{A}$ valid?

As an example application, consider a web application that takes user input (e.g., a username and password) and generates a query to a backend database (e.g., a banking system) to authenticate the user. Errors in the application may allow a malicious user to send specifically crafted input to cause the application to generate a query with a tautology as its conditional clause. This is one example of a widespread security vulnerability

* This research was supported in part by NSF CAREER Grant No. 0546844 and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

$$G = (\ \{\vee, \wedge, \neg, (,), -, +, =, \neq, >, \leq, <, \geq, \} \cup \Re \cup V,$$
$$\{\mathsf{bE}, \mathsf{bT}, \mathsf{bF}, \mathsf{bS}, \mathsf{pred}, \mathsf{aE}, \mathsf{aT}\},\ P_G,\ \mathsf{bE})$$

$$P_G = \left\{ \begin{array}{lcllcl} \mathsf{bE} & ::= & \mathsf{bE} \vee \mathsf{bT} \mid \mathsf{bT} & \mathsf{aE} & ::= & \mathsf{aE} + \mathsf{aT} \mid \mathsf{aT} \\ \mathsf{bT} & ::= & \mathsf{bT} \wedge \mathsf{bF} \mid \mathsf{bF} & \mathsf{aT} & ::= & V \mid -V \mid \Re \\ \mathsf{bF} & ::= & \neg\, \mathsf{bS} \mid \mathsf{bS} & \mathsf{cmp} & ::= & = \mid \neq \mid > \\ \mathsf{bS} & ::= & (\ \mathsf{bE}\ ) \mid \mathsf{pred} & & & \mid \geq \mid < \mid \leq \\ \mathsf{pred} & ::= & \mathsf{aE\ cmp\ aE} & & & \end{array} \right\}$$

Figure 1. Grammar G for linear arithmetic constraints, where V is a set of variables.

known as *database command injection* [1]. These vulnerabilities can be discovered statically by constructing a language generator $\mathcal{A}$ to conservatively characterize the set of database queries that the application may generate [2]. The verification problem then reduces to checking whether $\mathcal{A}$ accepts any tautologies.

We denote this class of problems parametrically as $\text{VALID}_{\Pi,\Phi,K}$. The first parameter, Π, is the formalism for describing the language generator $\mathcal{A}$ that $\text{VALID}_{\Pi,\Phi,K}$ takes as input. The second parameter, Φ, is the theory under which each string in $\mathcal{L}(\mathcal{A})$ (i.e., the language of $\mathcal{A}$) is to be interpreted. The third parameter, $K \in \{\exists, \forall\}$, specifies whether the goal is to find whether any ($K = \exists$) or all ($K = \forall$) constraints in $\mathcal{L}(\mathcal{A})$ are tautologies. This paper introduces such decision problems by presenting an algorithm for $\text{VALID}_{\text{FSA,LA},\exists}$, where "FSA" is short for "Finite State Automaton," and "LA" is short for "Linear Arithmetic." In practice, FSAs are sufficient for modeling web applications as query constructors [2].

The challenge of $\text{VALID}_{\Pi,\Phi,K}$ for any non-trivial Π is that $\mathcal{L}(\mathcal{A})$ may be infinite, so naively enumerating $\mathcal{L}(\mathcal{A})$ and checking each constraint will not yield a decision procedure. Instead, the algorithm must exploit the finiteness of the representation of $\mathcal{A}$.

The rest of the paper is structured as follows. Section 2 presents the $\text{VALID}_{\text{FSA,LA},\exists}$ problem more precisely and defines *arithmetic loops* and *logical loops*, which represent the main challenges of the problem. Sections 3 and 4 address arithmetic and logical loops respectively. Section 5 surveys related work, and Sect. 6 concludes.

2 Overview

This section first defines the parameters for $\text{VALID}_{\text{FSA,LA},\exists}$ and makes some general observations, and then sets up the high-level structure of the algorithm.

2.1 The $\text{VALID}_{\text{FSA,LA},\exists}$ Problem

Finite state automata (FSAs) are defined by a five-tuple, $(Q, \Sigma, \delta, q_0, q_f)$, where Q is a set of states, Σ is the alphabet of terminals from the input language, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $q_0 \in Q$ is a start state, and $q_f \in Q$ is a final state. The semantics of FSAs is standard. The grammar G in Fig. 1 defines the syntax for linear arithmetic constraints. Again, the semantics of the language is standard, and the grammar rules reflect the operator precedence. Because each $s \in \mathcal{L}(\mathcal{A})$ must be interpreted as a linear arithmetic constraint, for $\mathcal{A}$ to be a valid input to $\text{VALID}_{\text{FSA,LA},\exists}$, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(G)$. For the sake of compactness and for certain steps in our algorithm, the transition relation will sometimes be presented as $\delta \subseteq Q \times \Sigma^* \times Q$.

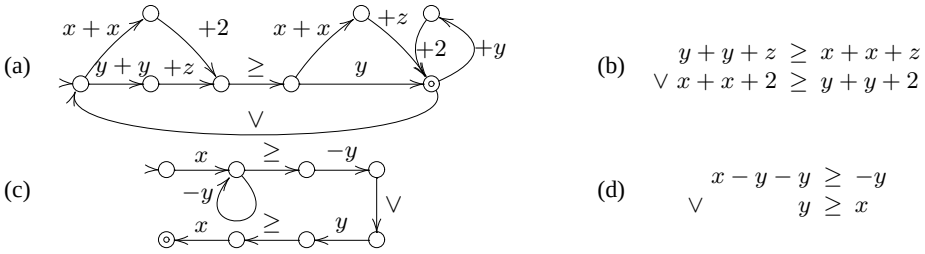

Figure 2. Two example FSAs.

$$\begin{array}{l} \text{let} \quad a_{ik} = (q_i, a, q_k) \quad b_{ij} = (q_i, b, q_j) \quad c_{jk} = (q_j, c, q_k) \\ \begin{array}{ll} a \rightarrow b\ c \in P_G & \\ b_{ij}, c_{jk} \in \delta_T \cup \delta_N & \Rightarrow \end{array} \begin{array}{l} \delta_N = \delta_N \cup \{a_{ik}\} \\ \delta_R(a_{ik}) = \delta_R(a_{ik}) \cup \{\{b_{ij}, c_{jk}\}\} \end{array} \end{array}$$

Figure 3. CFL-reachability algorithm—the cases for *rhs*'s of lengths other than 2 are analogous.

We select Φ = "LA" to explore because it is broadly applicable, and the general problem of validity checking for integer arithmetic constraints is undecidable (due to the undecidability of Diophantine equations [3]). Although multiplication by a constant is within the theory of linear arithmetic, we forbid '$\times$' from appearing in Σ. If we allowed, for example, an FSA to have a loop over "$\times 2$," we would characterize the multiplication as "$\times 2^n$," and exponentiation with variables is difficult to reason about.

A few concrete examples of inputs to $\text{VALID}_{\text{FSA,LA},\exists}$ help to illustrate the significance of the finite representation and the challenges in handling it. Consider, for example, the FSA shown in Fig. 2a. Because of cycles in the automaton, it accepts an infinite language. By considering single passes through each of its cycles, we discover the tautology shown in Fig. 2b. However, a single pass through a cycle is not sufficient to discover possible tautologies in general. For example, two passes through the cycle in the FSA shown in Fig. 2c are needed to discover the tautology in Fig. 2d.

Our algorithm for validity checking of automata uses a combination of automaton transformations and a theorem that bounds the number of constraints needed for a tautology. It generates validity queries in the theory of first-order arithmetic and sends them to a first-order arithmetic decision procedure [4]. If the FSA accepts some tautology, at least one of the finite number of first-order arithmetic queries must be a tautology.

2.2 Definitions and Setup

Our algorithm for the $\text{VALID}_{\text{FSA,LA},\exists}$ problem uses a modified version of context free language (CFL) reachability to create abstractions of the input FSA for use at certain steps. This CFL-reachability algorithm takes as input a context free grammar $G = (N, \Sigma, P_G, S)$ and an FSA $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_F)$, and produces an augmented FSA $\mathcal{A}' = (Q, \Sigma \cup N, \delta_T \cup \delta_N, \delta_R, q_0, q_F)$ where δ_T and δ_N are sets of *terminal transitions* and *non-terminal transitions* (transitions labeled with terminals and non-terminals from G) respectively, and $\delta_R : \delta_N \rightarrow \mathcal{P}(\mathcal{P}(\delta_N \cup \delta_T))$ is the set of *reference transitions*. The transitions in $\mathcal{A}'$ are defined by $\delta_T = \delta$ plus the minimal solution to the constraint shown in Fig. 3. The standard CFL-reachability algorithm does not include reference

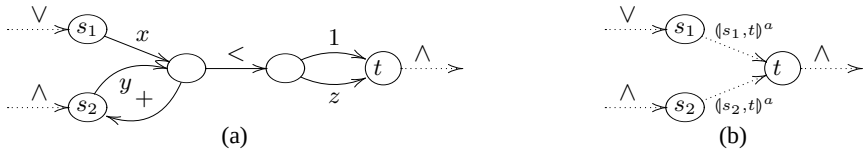

Figure 5. Examples for arithmetic and logical FSAs.

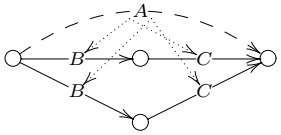

Figure 4. CFL-reachability.

transitions [5]. Figure 4 depicts an FSA produced by CFL-reachability, showing terminal transitions as solid, nonterminal transitions as dashed, and reference transitions as dotted, assuming that $A \rightarrow B\ C \in P_G$. For $t \in \delta_N$, we write $t \rightsquigarrow\!\!\gg t'$ if $t \in s_t \in \delta_R(t)$ for some s_t; let ' $\rightsquigarrow\!\!\gg^*$ ' denote the reflexive, transitive closures of ' $\rightsquigarrow\!\!\gg$,' and let $\delta^*_R(t) = \bigcup_{s_t \in \delta_R(t)}(s_t \cup \bigcup_{t' \in s_t} \delta^*_R(t'))$. The references effectively form parse trees for all of the strings in $\mathcal{L}(\mathcal{A})$—"all" because of the syntactic correctness requirement, i.e., $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(G)$.

Let "σ_{ij}" abbreviate "(q_i, σ, q_j)." Because bE cannot be derived from aE in R_G, if $\mathsf{aE}_{ij} \in \delta_N$, then for any string s accepted on a q_i–q_j path over the transitions in $\delta^*_R(\mathsf{aE}_{ij})$, $s \in \mathcal{L}(N, \Sigma, P_G, \mathsf{aE})$. Similarly, if $\mathsf{bE}_{ij} \in \delta_N$, then for any string s generated on a q_i–q_j path, $s \in \mathcal{L}(N, \Sigma, P_G, \mathsf{bE})$. This leads to the following lemma:

Lemma 1. *Each cycle in $\mathcal{A}$, an input to* VALID$_{FSA,LA,\exists}$*, is either arithmetic (i.e., within $\delta^*_R(\mathsf{aE}_{ij})$ for some $\mathsf{aE}_{ij} \in \delta_N$) or logical (i.e., within $\delta^*_R(\mathsf{bE}_{ij})$ for some $\mathsf{bE}_{ij} \in \delta_N$ and not within $\delta^*_R(\mathsf{aE}_{kl})$ for any $\mathsf{aE}_{kl} \in \delta_N$).*

The subsequent sections present one technique for handling arithmetic cycles and another for logical cycles, but neither technique works for both kinds of cycles. This motivates the primary CFL-reachability-based abstraction used in our algorithm.

Definition 1 (Arithmetic FSA). *Let $\mathcal{A} = (Q, \Sigma, \delta, \delta_R, q_0, q_F)$ and $\mathsf{pred}_{st} \in \delta$. The* arithmetic FSA $\langle\!|q_s, q_t|\!\rangle^a$ *or* $\mathcal{A}_{st} = (Q', \Sigma, \delta', \delta'_R, q_s, q_t)$ *where* $Q' = \{q \in Q \mid (q, \sigma, q') \in \delta'\} \cup \{q_t\}$, $\delta' = \{t \in \delta \mid t \in \delta'^*_R(\mathsf{pred}_{st})\}$, *and* $\delta'_R(t) = \delta_R(t)$ *if* $t \in \delta'$ *and* $\emptyset$ *otherwise.*

Definition 2 (Logical FSA). *Let $\mathcal{A} = (Q, \Sigma, \delta, \delta_R, q_0, q_F)$. The* logical FSA $\mathcal{A}_l = (Q', \Sigma \cup \mathcal{B}, \delta', \delta'_R, q_0, q_F)$, *where* $Q' = \{q \in Q \mid (q, \sigma, q') \in \delta'\} \cup \{q_F\}$, $\mathcal{B} = \{\langle\!|q_i, q_j|\!\rangle^a \mid q_i, q_j \in Q'\}$, $\delta' = \{\sigma_{ij} \in \delta \mid \sigma \neq \mathsf{pred} \wedge \sigma_{ij} \in \delta'^*_R(\mathsf{bE}_{0F})\} \cup \{\langle\!|q_i, q_j|\!\rangle_{ij} \mid \mathsf{pred}_{ij} \in \delta\}$, $\delta'_R(t \notin \delta') = \emptyset$, *and* $\delta'_R(t \in \delta') = \bigcup_{s_t \in \delta_R(t)}(\{\{\langle\!|q_i, q_j|\!\rangle^a_{ij}\}\}$ *if* $s_t = \{\mathsf{pred}_{ij}\}$ *;* $\{s_t\}$ *otherwise*).

The FSA fragment in Fig. 5a has two arithmetic FSAs. The one defined by $\langle\!|s_1, t|\!\rangle^a$ includes all states and solid transitions in the figure. The one defined by $\langle\!|s_2, t|\!\rangle^a$ excludes the state s_1 and the x-transition. Figure 5b shows the logical FSA that results from abstracting out the arithmetic FSAs in Fig. 5a. The labels on the states show the correspondence between the original FSA and the logical FSA. Arithmetic FSAs include no logical cycles and logical FSAs include no arithmetic cycles.

We split the problem of validity for FSAs into two subproblems, and in order to define the sub-problems precisely, we define *linear FSAs*:

Definition 3 (Linear FSA). *An FSA $\mathcal{A}$ is a* linear FSA *iff $\mathcal{A}$ is deterministic, $|\mathcal{L}(\mathcal{A})| = 1$, and $\mathcal{A}$ is minimal (i.e., it includes no useless states or transitions).*

The first subproblem takes as input a linear logical FSA and produces a linear arithmetic constraint that is valid if the linear logical FSA accepts a tautology. To this end, Sect. 3 casts arithmetic FSAs as network flow problems. The second subproblem takes as input a logical FSA and produces a finite number of linear logical FSAs such that at least one accepts a tautology iff the input FSA accepts a tautology. Section 4 uses a finite model theorem to unroll logical loops based on the number of variables in the arithmetic FSAs.

3 Arithmetic Loops

We address arithmetic loops by casting questions about arithmetic automata as questions about network flows. The algorithm has four main steps. First, given an arithmetic FSA $\mathcal{A} = (Q, \Sigma, \delta = (\delta_T \cup \delta_N), \delta_R, q_s, q_t)$ we define a labelling function

$$L : \Big(\delta \ \cup \bigcup_{\substack{t \in \delta \\ s_t \in \delta_R(t)}} (t, s_t)\Big) \to \mathcal{F}$$

where $\mathcal{F}$ is a set of *flow variables*. In the constraint that this construction generates, the value of $L(t \in \delta_T)$ equals the number of times the transition t was taken in some accepting path. The value of $L(t, s_t)$ equals the number of times the corresponding derivation occurs in the parse tree of the generated string. The first part of the constraint existentially quantifies the flow variables because the $\text{VALID}_{\text{FSA,LA},\exists}$ problem asks whether *there exist* any tautologies: "$\exists_{f \in \text{codom}(L)} f$."

The second step constrains the values of flow variables so that the values they can take correspond to derivations and paths through $\mathcal{A}$.

$$(1) \bigwedge_{f \in \text{codom}(L)} f \geq 0 \qquad \wedge \qquad (2) \bigwedge_{t \in \delta_N} L(t) = \sum_{s_t \in \delta_R(t)} L(t, s_t) \qquad \wedge$$

$$(3) \bigwedge_{t \in \delta} L(t) = k + \sum_{\substack{t' \in \delta \\ s_{t'} \in \delta_R(t') \\ t \in s_{t'}}} L(t', s_{t'}) \ , \ k = \begin{array}{l} 1 \text{ if } t = \mathsf{pred}_{st} \\ 0 \text{ otherwise} \end{array}$$

Conjunction (1) prohibits solutions that would have a transition being traversed a negative number of times. Figure 6 illustrates how (2) and (3) balance the flow of incoming and outgoing reference transitions.

The third step universally quantifies the variables in $V \cap \Sigma$ because the $\text{VALID}_{\text{FSA,LA},\exists}$ problem asks whether there exists a *tautology*: "$\forall_{v \in V \cap \Sigma} v$."

The fourth step uses $\text{C}(\mathsf{pred}_{st})$ to generate a *flow-comparison* constraint that relates the number of times

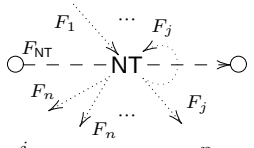

Figure 6. Flow balancing.

each transition is taken with the value of the generated expression. Because addition commutes, C uses the number of times each term occurs to calculate the value of arithmetic expressions.

$$\mathrm{C}(\mathsf{pred}_{st}) = \bigwedge_{\substack{\{\mathsf{aE}_{si},\mathsf{cmp}_{ij},\mathsf{aE}_{jt}\}\in\delta_R(\mathsf{pred}_{st})\\ \{c_{ij}\}\in\delta_R(\mathsf{cmp}_{ij})}} (L(c_{ij}) = 1) \Rightarrow \mathrm{C}(\mathsf{aE}_{si})\ c\ \mathrm{C}(\mathsf{aE}_{jt}) \qquad \mathrm{C}(\mathsf{aE}_{ij}) = \sum_{\mathsf{aE}_{ij} \rightsquigarrow^{*} \mathsf{aT}_{kl}} \mathrm{C}(\mathsf{aT}_{kl})$$

$$\mathrm{C}(\mathsf{aT}_{ij}) = \sum_{\{v_{ij}\}\in\delta_R(\mathsf{aT}_{kl})} (L(\mathsf{aT}_{ij}, \{v_{ij}\}) \times v) \quad - \sum_{\{-_{ik},v_{kj}\}\in\delta_R(\mathsf{aT}_{kl})} (L(\mathsf{aT}_{ij}, \{-_{ik}, v_{kj}\}) \times v)$$

Tarski's theorem [4] establishing the decidability of first-order arithmetic guarantees that expressions of this form are decidable when the variables range over real numbers. We state here a completeness result:

Theorem 1. *If the flow-comparison expression generated from an arithmetic FSA $(\!|s, t|\!)$ is not valid, then $(\!|s, t|\!)$ does not accept a tautology.*

Furthermore, when two or more arithmetic FSAs are linked sequentially by logical operators (e.g., '$\wedge$' or '$\vee$'), we can merge in a natural way the constraints that model the arithmetic FSAs, and the completeness result holds for the sequence of automata.

Theorem 2. *If the flow-comparison expression generated from a linear logical FSA $\mathcal{A}$ is not valid, then $\mathcal{A}$ does not accept a tautology.*

Unsoundness If the flow variables ranged over integers, this construction would be sound. Because they range over real numbers, they may take on non-integral values and not correspond to any path through the FSA.

4 Logical Loops

Consider an arithmetic FSA with an $\vee$-transition from its last state to its first state. The arithmetic FSA might not accept any tautology, but two or more passes through the arithmetic FSA joined by '$\vee$' may be a tautology, as in the case of the FSA in Fig. 2a.

4.1 Setup and Loop Unrolling

Unfortunately, we cannot use equations to address logical loops as we did for arithmetic loops. If we did, the generated constraint would not be in first-order arithmetic. Instead, we "unroll" the loop a bounded number of times so that if the loop accepts some tautology, the unrolling must also accept some tautology.

The algorithm for generating linear logical FSAs from a given logical FSA has three main steps. (1) Remove the $\neg$-transitions. (2) Collapse all strongly connected components (SCCs) in the FSA to form a dag, and enumerate the paths through the dag. (3) Transform each collapsed SCC in an FSA $\mathcal{A}_l$ into a linear FSA that replaces the SCC in $\mathcal{A}_l$.

The first step uses graph transformations and an adaptation of DeMorgan's law to propagate '$\neg$' inward. Due to space constraints, the details are omitted here but can be

found in the companion technical report [6]. The second step is straightforward, so we omit the details. Section 4.2 presents the third step in detail. Each step preserves the property of accepting a tautology.

The third step takes as input a logical FSA without $\neg$-transitions that only produces syntactically correct strings (as stated in Sect. 2.2). This step relies on the syntactic correctness property, which implies that parentheses are balanced on all paths, and the parenthetic nesting depth is bounded. Because parentheses are always balanced, we can abstract all paths between a pair of matching parentheses into a *parenthetic FSA* $(\!|q_i, q_j|\!)^p$, which is defined as follows:

Definition 4 (Parenthetic FSA). *Let logical FSA* $\mathcal{A} = (Q, \Sigma, \delta, \delta_R, q_0, q_F)$ *where* $\mathsf{bS}_{st} \in \delta$ *and* $\mathsf{bS}_{st} \rightsquigarrow^* \mathsf{bE}_{kl}$. *The* parenthetic FSA $(\!|q_s, q_t|\!)^p$ *or* $\mathcal{A}_{st} = (Q', \Sigma, \delta', \delta'_R, q_s, q_t)$ *where*

$$
\begin{aligned}
Q' &= \{q \in Q \mid (q, \sigma, q') \in \delta'\} \cup \{q_t\} \\
\mathcal{B}_p &= \{(\!|q_i, q_j|\!)^p \mid q_i, q_j \in Q'\} \\
\delta' &= \{t \in \delta \mid \neg\exists \mathsf{bS}_{ij}.\, \mathsf{bS}_{st} \rightsquigarrow^+ \mathsf{bS}_{ij} \rightsquigarrow^+ t\} \\
&\quad \cup \{(\!|q_i, q_j|\!)^a_{ij} \in \delta \mid \mathsf{bS}_{ij} \in \delta'\} \\
&\quad \cup \{(\!|q_i, q_j|\!)^p_{ij} \mid \mathsf{bS}_{ij} \in \delta' \wedge \mathsf{bS}_{ij} \rightsquigarrow^* \mathsf{bE}_{kl}\} \\
\delta'_R(t \notin \delta') &= \emptyset \\
\delta'_R(t \in \delta') &= \begin{cases} \{\{(\!|q_i, q_j|\!)^a_{ij}, (\!|q_i, q_j|\!)^p_{ij}\} \cap \delta'\} & \text{if } \mathsf{bS}_{ij} = t \neq \mathsf{bS}_{st} \\ \delta_R(t) & \text{otherwise} \end{cases}
\end{aligned}
$$

This abstraction is analogous to the abstraction that defines arithmetic and logical FSAs, except that parenthetic FSAs can be nested within parenthetic FSAs.

After abstracting away parenthetic FSAs, the SCC only has $\vee$- and $\wedge$-transitions and transitions over arithmetic and parenthetics FSAs as atomic units. The following theorem provides the basis for the bounded loop unrolling in Sect. 4.2:

Theorem 3 (Loop Unrolling Theorem). *Let* $T = \{t_1, \cdots, t_m\}$, *where each* t_i *is a comparison of linear arithmetic expressions, and let* n *be the number of distinct variables in* T. *Then* $(\bigvee_{t \in T} t)$ *is a tautology iff there exists some* $T' \subseteq T$ *such that* $|T'| \leq (n+2)$ *and* $(\bigvee_{t \in T'} t)$ *is a tautology.*

Proof. Helly's theorem states that if $K_1, \cdots, K_m$ are convex sets in n-dimensional Euclidean space $\Re^n$ in which $m \geq n$, and if for every choice of $n+1$ of the sets K_i there exists a point that belongs to all the chosen sets, then there exists a point that belongs to all the sets $K_1, \cdots, K_m$ [7]. This implies that if $K_1, \cdots, K_m$ are convex sets as before but have no points in common, then there exists some choice of $n+1$ of the sets K_i that have no points in common.

If $t_1 \vee \cdots \vee t_m$ is a tautology, then by DeMorgan's law, $\neg t_1 \wedge \cdots \wedge \neg t_m$ is unsatisfiable. Each $\neg t_i$ can be rewritten as s_i by replacing the comparison operator with its opposite (e.g., $< \rightleftarrows \geq$). Linear inequalities and linear equalities each define convex spaces (half-spaces and hyperplanes, respectively) in $\Re^n$, where n is the number of variables occurring in them. If each s_i falls into one of these categories (i.e., its comparison operator is one of $\{<, >, \geq, \leq, =\}$), then some choice of $n+1$ of them is unsatisfiable, and the disjunction of the corresponding t_i's is a tautology.

$$
\begin{aligned}
\mathbf{Paren}(\{s_1,\ldots,s_n\}) &= \mathbf{Conj}(s_1) \vee \cdots \vee \mathbf{Conj}(s_n)\\
\mathbf{Conj}(\{b_1,\ldots,b_n\}) &= \mathbf{(}\,\mathbf{Disj}(b_1)\,\mathbf{)} \wedge \cdots \wedge \mathbf{(}\,\mathbf{Disj}(b_n)\,\mathbf{)}\\
\mathbf{Disj}(\langle\!|q_i,q_j|\!\rangle^p_{ij}) &= \mathbf{Paren}(\textstyle\bigcup_{t\in\{\mathsf{bT}_{kl}\in\delta_N \mid \mathsf{bE}_{kl}\in\delta^*_R(\mathsf{bS}_{ij})\}} \mathbf{Paths}(t,\emptyset))\\
\mathbf{Disj}(\langle\!|q_i,q_j|\!\rangle^a_{ij}) &= \underbrace{\langle\!|q_i,q_j|\!\rangle^a \vee \cdots \vee \langle\!|q_i,q_j|\!\rangle^a}_{NumVars+2}\\
\mathbf{Paths}(\ t\in\delta_N,\ d\ ) &= \textstyle\bigcup_{s_t\in\delta_R(t)} \mathbf{Zip}(s_t \setminus (d\cup\{t\}), d\cup\{t\})\\
\mathbf{Paths}(\langle\!|q_i,q_j|\!\rangle^a_{ij}, d) &= \{\{\langle\!|q_i,q_j|\!\rangle^a_{ij}\}\\
\mathbf{Paths}(\langle\!|q_i,q_j|\!\rangle^p_{ij}, d) &= \{\{\langle\!|q_i,q_j|\!\rangle^p_{ij}\}\}\\
\mathbf{Paths}(\ \wedge_{ij}\ ,\ d\ ) &= \emptyset\\
\mathbf{Zip}(\{t_1,\ldots,t_n\}, d) &= \textstyle\bigcup_{\substack{s_1\in\mathbf{Paths}(t_1,d)\\ \vdots\\ s_n\in\mathbf{Paths}(t_n,d)}} \{\textstyle\bigcup_{i=1}^n s_i\}
\end{aligned}
$$

Figure 7. Algorithm to construct a linear FSA from an SCC of a logical FSA.

However, a linear disequality (i.e., $\boldsymbol{a}\cdot\boldsymbol{x}\neq b$) defines a non-convex region. Specifically, the points that do not satisfy a disequality lie in a single hyperplane. Suppose that for all choices of $n+1$ convex regions (as defined by the s_i's) there exists some point p that satisfies the s_i's. Suppose further that for some choice of $n+1$ convex regions all points common to the region lie in the hyperplane that does not satisfy some disequality s_i. Then a choice of $n+2$ of the s_i's are required for unsatisfiability, and consequently $n+2$ of the t_i's are required for validity.

The only non-convex shape definable by linear constraints has a hyperplane as its region of unsatisfiability. Consequently, given a set S of convex regions whose intersection (is necessarily convex and) is not confined to a hyperplane, no addition of a finite number of non-convex linear constraints to S can cause S to become unsatisfiable. Therefore, no more than $(n+2)$ t_i's will be needed for a tautology. □

4.2 Linearizing Strongly Connected Components

Figure 7 defines five functions: **Paren**, **Conj**, **Disj**, **Paths**, and **Zip**. These five functions are used to construct a linear logical FSA $\mathcal{A}_l$ from a strongly connected component of a logical FSA $\mathcal{A}_s$ such that $\mathcal{A}_l$ accepts a tautology iff $\mathcal{A}_s$ accepts a tautology.

The algorithm to construct $\mathcal{A}_l$ begins as follows. Let $\mathcal{A}_s$ be an SCC without $\neg$-transitions, with parenthetic FSAs abstracted, and with start and final states q_0 and q_F, respectively. Construct a single parenthetic FSA by adding to δ $(q_\alpha, \mathbf{(}, q_0)$ and $(q_F, \mathbf{)}, q_\beta)$ and letting q_α and q_β be the start and final states, respectively. Begin constructing a linear FSA by calling $\mathbf{Disj}(\langle\!|q_\alpha,q_\beta|\!\rangle^p_{\alpha\beta})$. **Disj** interprets $(\langle\!|q_\alpha,q_\beta|\!\rangle^p_{\alpha\beta})$ as the parenthetic FSA that it represents. The set $\{\mathsf{bT}_{kl}\cdots\}$ over which $\mathbf{Disj}()$ takes a union includes all of the nonterminal transitions from which only conjunctive expressions (i.e., "$a\wedge\cdots\wedge b$") can be derived, but all expressions the can be derived can be entered and exited through $\vee$-transitions. The **Paths** function then returns a set S of sets of transitions, where each set s of transitions includes all of the arithmetic and parenthetic

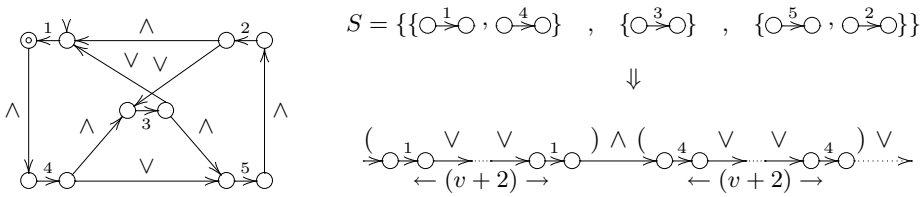

Figure 8. An example SCC and the result of Fig. 7, where $v = NumVars$.

FSAs on some shortest (i.e., $\neg\exists s' \in S.s' \subset s$) acyclic path derived from the transition t. Because $\mathcal{A}_s$ is strongly connected, each path represented by the set returned from **Paths** can be entered and exited through '$\vee$,' and disjunction weakens expressions monotonically, if a tautology can be constructed from the conjunctive expressions returned from **Paths**, then $\mathcal{A}_s$ accepts a tautology. Because conjunction strengthens expressions monotonically, and the conjunctive expressions returned from **Paths** are all **Paths** returns all shortest conjunctive expressions, if $\mathcal{A}_s$ accepts some tautology, then a tautology can be constructed by the shortest conjunctive expressions.

Paths passes the set representing all shortest conjunctive expressions to **Paren**, which begins to construct a linear structure by calling **Conj** on each expression, and connecting the results with '$\vee$.' A DNF expression constructed by disjoining several instances of one of these conjunctive expressions can be refactored into a CNF expression. **Conj** constructs such a CNF expression. Because it constructs a CNF expression, each element (arithmetic or parenthetic FSA) in the set can be handled individually and independently by **Disj**. If the element is a parenthetic FSA, **Disj** recurses down and produces a linear construction based on the parenthetic FSA. If the element is an arithmetic FSA, **Disj** disjoins $NumVars + 2$ copies of it, where $NumVars$ is the number of distinct variables that appear in the original FSA (i.e., $|\{v \in V \mid v_{ij} \in \delta\}|$). Theorem 3 implies that if any (necessarily finite) disjunction of constraints from a given set constitutes a tautology, then at most $NumVars + 2$ of the constraints are needed to construct a tautology. Given a complete linear structure, a linear logical FSA can be constructed by using the sequence of tokens as the labels for the transitions in a linear FSA.

To illustrate the algorithm, Fig. 8 shows an example SCC, where numbers (1–5) represent arithmetic FSAs. The set S consists of three sets, and below that, the beginning of the constructed linear FSA shows how those sets are used. Note that each set in S consists of the arithmetic FSAs along a path that can be entered and exited from $\vee$-transitions but has only $\wedge$-transitions between arithmetic FSAs.

4.3 Soundness, Completeness, and Complexity

Taken together, the algorithms for constructing linear logical FSAs from a logical FSA are sound and complete for finding tautologies in logical FSAs.

Theorem 4 (Soundness and Completeness). *Given an FSA $\mathcal{A}$ where $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(G)$, the algorithm presented in Sect. 4 produces a finite set $S_{\mathcal{A}}$ of linear logical FSAs such that there exists an FSA in $S_{\mathcal{A}}$ that accepts a tautology iff $\mathcal{A}$ accepts a tautology.*

Proof. The abstraction from $\mathcal{A}$ to a logical FSA $\mathcal{A}'$ described in Sect. 2.2 maintains language equivalence if a path through $\mathcal{A}'$ includes paths through the arithmetic FSAs that correspond to their names. The algorithm to remove $\neg$-transitions produces a logical FSA $\mathcal{A}^+$ from $\mathcal{A}'$ such that there exists a bijective mapping $b : \mathcal{L}(\mathcal{A}^+) \rightarrow \mathcal{L}(\mathcal{A}')$ where $b(\varphi^+) = \varphi'$ implies $\varphi^+ \equiv \varphi'$. Collapsing SCCs and finding paths through the dag produces a finite set $S_{\mathcal{A}^+}$ of FSAs from $\mathcal{A}^+$ such that a path in $\mathcal{A}^+$ can be mapped to a path in some $\mathcal{A}_S \in S_{\mathcal{A}^+}$, and vice versa. The algorithm in Fig. 7 then produces a finite set $S_{\mathcal{A}}$ of linear logical FSAs from $S_{\mathcal{A}^+}$, such that, by the algorithm in Sect. 4.2 and Theorem 3, there exists an FSA $\mathcal{A}^- \in S_{\mathcal{A}}$ that accepts a tautology iff some $\mathcal{A}_S \in S_{\mathcal{A}^+}$ accepts a tautology. □

A logical FSA is no larger than the FSA from which it is abstracted. Removing $\neg$-transitions produces at most a constant number of instances of each state, so the resulting FSA has size $O(|Q|)$ in the size of the input. The states in the FSA can be partitioned into Q_q, those states that can be reached from themselves, Q_p, those that cannot. Let $q = |Q_q|$ and $p = |Q_p|$. Collapsing SCCs and enumerating all paths generates $O(2^p)$ paths. From each of these paths, the algorithm in Fig. 7 produces linear logical FSAs. If $p \gg q$, then each path has length $O(p)$. Otherwise, each path is bounded by $|S|$, which is $O(2^q)$, and the length of the FSA produced from each $S \in \int$, which is $O(n(q2^q))$, where n is the number of unique variables in the arithmetic FSAs. So, each path has length $O(\max(p, nq2^q))$. From each path a query, which is linear in the size of the path, is created and sent to a first-order arithmetic decision procedure.

5 Related Work

This section surveys closely related work.

First-Order Theories Tarski established the decidability of the first-order theory of real numbers with addition and multiplication through quantifier elimination [4]. Collins used cylindrical decomposition to check validity in the same theory more efficiently, but his algorithm also has high complexity [8]. The first-order theory over integers is undecidable because of the undecidability of solving Diophantine equations [3]. However, an important fragment, Presburger Arithmetic, is decidable [9].

Linear Constraints In program analysis and formal verification, decision procedures for linear constraints are widely used. Some proposed techniques include Fourier-Motzkin variable elimination [10], the Sup-Inf method of Bledsoe [11], and Nelson's method based on Simplex [12]. More tractable algorithms can be found by restricting the class of integer constraints further. Pratt gives a polynomial time algorithm for the form of linear constraints $x \leq y + k$, where k is an integer [13]. Shostak considers a slightly more general problem $ax+by \leq k$, where a, b, and k are integer constants [14]. He uses "loop residues" for an algorithm which requires exponential time in the worst case. Aspvall and Shiloach give a refined algorithm for the same form which runs in polynomial time [15]. Su and Wagner [16] leverage ideas from Pratt and Shostak to propose the first polynomial time algorithm for a general class of integer range constraints underlying the standard example (range constraints [17]) of abstract interpretation [18].

Combined Theories In 1979, Nelson and Oppen proposed a method for combining theories in a decision procedure [19]. Contemporary theorem provers, such that as in Necula and Lee's certifying compiler [20], use Nelson and Oppen's architecture for cooperating decision procedures. In 1984, Shostak introduced an algorithm for deciding the satisfiability of quantifier-free formulas in a combined theory [21]. This algorithm improved over previous decision procedures by enabling multiple theories to be integrated uniformly instead of using separate, communicating processes. This algorithm serves as the basis for decision procedures found in several tools including PVS [22], STeP [23], and SVC [24]. SVC uses a decision procedure for a fragment of first-order logic which excludes quantifiers, but includes equality, uninterpreted functions and constants, arrays, records, and bit-vectors, as well as propositional connectives. CVC Lite [25] is a descendant of SVC that includes a builtin SAT solver and support for quantifiers.

Helly-like Theorems Helly-like theorems have been used to improve certain individual linear programming problems, such as finding a point in the intersection of a family of convex sets [7, 26]. In 1994, Amenta proved a general relation between Helly-like theorems and generalized linear programming [27]. None of these, however, use Helly's theorem as this paper does: to bound the number of constraints needed to find a tautology in unboundedly large sets of constraints.

6 Conclusions and Future Work

We have introduced the class of decision problems for language generators $\text{VALID}_{\Pi,\Phi,K}$ (motivated by the need for advanced checking of meta-programs) and an algorithm for $\text{VALID}_{\text{FSA,LA},\exists}$. Our algorithm is based on casting FSAs as network flow problems and leveraging a novel application of Helly's theorem to bound the number of comparison expressions needed for a tautology. The network flow-based construction is unsound because the flow variables may take on non-integral values.

This paper opens up several interesting directions for future work. First, language generators that can match calls and returns, such as tree automata and push-down automata, are better suited for certain program analysis problems in meta-programming than finite state automata. Because the algorithm presented here relies on the boundedness of parenthetic nesting, new insights will be needed to construct algorithms over these more expressive formalisms. Second, we expect that similar techniques to the ones presented here will yield an algorithm for $\text{VALID}_{\text{FSA,LA},\forall}$. Third, this algorithm does not exploit much of the finer-grained structure of the FSA. We expect that this can be used to provide an alternative, and frequently lower, bound on the number of expressions needed for a tautology. Fourth, we are interested in studying the relation of $\text{VALID}_{\text{FSA,LA},\exists}$ to MSO logic, which also has an automata-based formulation. Finally, we would like to find matching upper and lower bounds for the $\text{VALID}_{\text{FSA,LA},\exists}$ problem in order to know its exact complexity.

References

1. Borland, M.: Advanced SQL Command Injection: Applying defense-in-depth practices in web-enabled database applications (2002)
2. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Proc. SAS'03. (2003) 1–18 URL: `http://www.brics.dk/JSA/`.
3. Matiyasevich, Y.: Solution of the tenth problem of Hilbert. Mat. Lapok **21** (1970) 83–87
4. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press (1951)
5. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: Proc. ICSE'04. (2004)
6. Wassermann, G., Su, Z.: Validity Checking for Finite Automata over Linear Arithmetic. Technical report, University of California, Davis, Computer Science Dept. (2006)
7. Danzer, L., Grünbaum, B., Klee, V.: Helly's theorem and its relatives. In: Proceedings of the Symposium on Pure Mathematics. Volume 7 of Convexity., AMS (1963) 101–180
8. Collins, G.E.: Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: A. Theory and Formal Languages. (1975)
9. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In: SAS, Springer-Verlag (1995) 21–32
10. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proc. Supercomputing. (1991) 4–13
11. Bledsoe, W.: The Sup-Inf method in Presburger arithmetic. Technical report, University of Texas Math Department (1974)
12. Nelson, G.: Techniques for program verification. Technical report, Xerox PARC (1981)
13. Pratt, V.: Two easy theories whose combination is hard. Technical report, MIT (1977)
14. Shostak, R.: Deciding linear inequalities by computing loop residues. J. ACM **28** (1981)
15. Aspvall, B., Shiloach, Y.: A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. SIAM Computing **9** (1980) 827–845
16. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In: Proc. TACAS'04. (2004)
17. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Symposium on Programming. (1976) 106–130
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 234–252
19. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. TOPLAS **1** (1979) 245–257
20. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: Proc. PLDI. (1998)
21. Shostak, R.E.: Deciding combinations of theories. J. ACM **31** (1984) 1–12
22. Owre, S., Shankar, N., Rushby, J.: PVS: A Prototype Verification System. In: Proc. CADE 11. (1992)
23. Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H., Uribe, T.E.: STeP: Deductive-algorithmic verification of reactive and real-time systems. In: Proc. CAV. (1996)
24. Barrett, C.W., Dill, D.L., Levitt, J.R.: Validity Checking for Combinations of Theories with Equality. In: Proc. FMCAD. (1996) 187–201
25. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Proc. CAV. (2004)
26. Avis, D., Houle, M.E.: Computational aspects of Helly's theorem and its relatives. International Journal of Computational Geometry Applications **5** (1995) 357–367
27. Amenta, N.: Helly-type theorems and generalized linear programming. Discrete & Computational Geometry **12** (1994) 241–261