Validity Checking for Finite Automata over Linear Arithmetic Constraints*

Gary Wassermann and Zhendong Su

University of California, Davis**

Abstract In this paper, we introduce a new validity checking problem over linear arithmetic constraints and present a decision procedure for the problem. Instead of considering the validity of any particular linear arithmetic constraint, we consider the following problem: Given a finite automaton accepting linear arithmetic constraints, does the automaton produce any constraint that is a tautology? This problem arises in the context of static verification of meta-programs, *i.e.*, programs dynamically generating other programs. This paper gives the first decision procedure to perform validity checking of finite automata over linear arithmetic constraints. Our algorithm will enable advanced verification of meta-programs.

1 Introduction

Many program analysis and formal verification problems reduce to validity or satisfiability checking over some logical theories. Consequently, significant effort has been put into designing efficient decision procedures for these theories. The validity of a given formula has been considered thus far in the literature. In this paper, we explore a new direction for the study of decision procedures. Rather than checking the validity of a given formula, we are interested in checking whether any formula accepted by a given finite state automaton (FSA) is valid. In particular, we consider the theory of linear arithmetic constraints, one of the most widely-used formalisms in program analysis and verification. The problem we study is: Given a finite automaton $\mathcal{A}$ accepting linear arithmetic constraints, does $\mathcal{A}$ accept a constraint ϕ such that ϕ is valid?

This problem arises in the static verification of *meta-programs*, programs manipulating and dynamically generating other programs. We motivate the problem through a concrete example in the analysis of web and database applications. Consider a web application that takes user input (*e.g.*, username and password) and dynamically constructs database queries to send to a backend database (*e.g.*, a banking system) to authenticate the user's access to the database. 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 known as *database command injection*. Here is how we can provide formal guarantee that a web application is free of this class of errors: By using an FSA to conservatively characterize the set of database queries a web application may generate, we

* Part of the results in this paper appeared in Workshop on Specification and Verification of Component-Based Systems (SAVCBS'04), a workshop with no formal proceedings.

** Authors' address: Department of Computer Science, 2063 Kemper Hall, University of California, Davis,CA 95616, USA, {wassermg,su}@cs.ucdavis.edu.

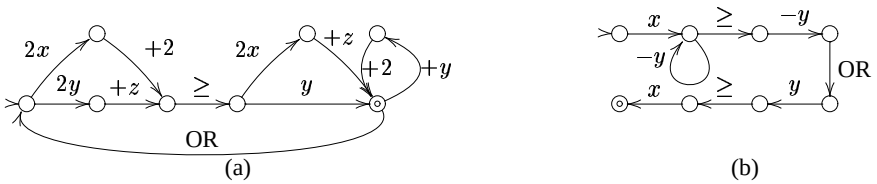


Figure 1. Two example FSAs

reduce the verification problem to the question of whether the FSA accepts any tautologies. We expect our decision procedure to be general and applicable in the checking and verification of other classes of meta-programs.

The challenge in validity checking for automata is that automata may produce infinite number of individual constraints. Consider, for example, the FSA shown in Fig. 1a, where "OR" denotes the logical operator $\vee$. Because of cycles in the automaton, it accepts an infinite language. A decision procedure must explore the automaton's finite structure. In the case of Fig. 1a, by considering single passes through each of its cycles, we discover the tautology "`2y + z ≥ 2x + z OR 2x + 2 ≥ y + y + 2`." However, a single pass through a cycle is not sufficient to discover possible tautologies in general. For example, two passes through the cycle in Fig. 1b are needed to discover the tautology "`x - y - y ≥ -y OR y ≥ x`."

Our algorithm for validity checking of automata uses a combination of automaton transformations and a novel theorem bounding the number of constraints needed for a tautology. Validity queries are generated and fed to a first-order arithmetic decision procedure. If the FSA accepts some tautology, at least one of the finite number of first-order arithmetic queries must be a tautology.

We assume that the FSA only generates syntactically correct linear arithmetic constraints. This assumption can also be enforced. The only difficulty is that in order to balance parentheses, a context free grammar is required to describe the syntax of linear arithmetic constraints. It is undecidable to check the inclusion of a regular language by a context-free language. However, we can check the automaton against some suitably close finite state under-approximation of the context-free grammar, as is done by Christensen *et al.* in a string analysis for Java [6].

We classify cycles as either *arithmetic* or *logical*, depending on the sort of expressions they accept. We conceptually view arithmetic portions of the FSA as network flow problems, and solve them using a decision procedure for first-order arithmetic. Logical loops cannot be handled this way. Instead, we bound the number of times they must be "unrolled" to ensure that if any tautology is accepted, at least one will be found.

Because of the undecidability of Diophantine equations [9], the general problem of validity checking for arithmetic constraints is undecidable. We restrict ourselves in this paper to discovering tautologies in linear arithmetic ("$+$" and "$-$" but no "$\times$") over real numbers. Multiplication by a constant is within linear arithmetic, and we allow it to appear in an acyclic region of the FSA. If we allowed, for example, an FSA to have a loop (cycle) over "$\times\ 2$," we would characterize the multiplication as "$\times\ 2^n$." Exponentiation with variables is difficult to reason about, so we exclude multiplication from cyclic regions of the FSA.

The rest of the paper is structured as follows. Section 2 provides some useful definitions based on the syntactic correctness requirement. Sections 3 and 4 address arithmetic and logical loops respectively, and Sect. 5 gives the complexity of our algorithm. Section 6 gives the proof of our key theorem, and Sect. 7 surveys some related work. Finally, Sect. 8 concludes.

2 Overview of Our Approach

In formulating an analysis to discover tautologies in the presence of cycles in an FSA, we first note a useful consequence of the syntactic-correctness property: the transitions of the FSA can be partitioned into four *transition types*. A transition of type:

(I) accepts part of an arithmetic expression ({`+`, `-`, `(`, `)`, `1`, `x`, ...}) before a comparison operator;
(II) accepts a comparison operator ($\{>, \geq, =, \leq, <, \neq\}$);
(III) accepts part of an arithmetic expressions after a comparison operator;
(IV) accepts a logical operator ({`AND`, `OR`, `NOT`}) or a parenthesis at the logical level.

This partitioning must be possible because, for example, if a transition that accepts a constant could be classified as both type I and type III, then the FSA would accept some string in place of a comparison expression which either had two comparison operators (*e.g.*, "...`x > 5 < 5`..") or none (*e.g.*, "...`AND 5 OR`..."). Consider also the notion of *parenthetic nesting* for each transition in a path as the number of logical (arithmetic) open parentheses minus the number of logical (arithmetic) closed parentheses encountered since the beginning of the path. Although a transition may be encountered on many different paths, it will always have the same parenthetic nesting. If this were not so, the FSA would accept some string with imbalanced parentheses.

Our analysis relies on this partitioning. Rather than trying to handle arbitrary cycles in the FSA uniformly, we classify each cycle as either *arithmetic*, if it only includes type I or type III transitions, or *logical*, if it includes type IV transitions. In order to handle each class of cycles without concerning ourselves with the other, we define an *arithmetic FSA* such that it can be viewed in isolation when we address arithmetic cycles and it can be abstracted out when we address logical cycles:

- The start state s immediately follows a type IV transition and immediately precedes a type I transition;
- The final state t immediately follows a type III transition and immediately precedes a type IV transition;
- All states and transitions are included that are reachable on some s-t path that has no type IV transitions.

The FSA fragment in Figure 2 has two arithmetic FSAs. The one defined by (s_1, t) includes all states and solid transitions in the figure. The one defined by (s_2, t) excludes the state s_1 and the x-transition. Finding the arithmetic FSAs in an FSA is straightforward through a simple graph traversal.

For reasoning about comparison expressions, which arithmetic FSAs accept, we view arithmetic FSAs as network flow problems with single source and sink nodes and solve these problems using a construction in linear arithmetic (see Section 3). Boolean

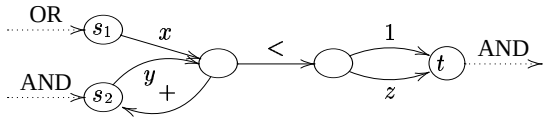
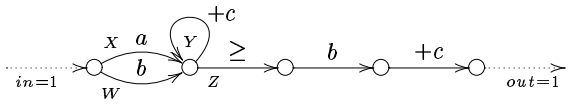


Figure 2. Example for arithmetic FSAs.



$\exists W, X, Y, Z:$ } fbw variables

$(1 = X + W) \wedge (X + W + Y = Y + Z) \wedge (Z = 1)$ } fbw balance equations

$\forall a, b, c:$ } arithmetic variables

$W \times (a) + X \times (b) + Y \times (c) \geq Z \times (b + c)$ } fbw-comparison expression

Figure 3. Flow equations for arithmetic loops.

expressions are comparison expressions connected with logical operators (*e.g.*, "AND," "OR," "NOT"). We discover tautologies by unrolling logical loops a bounded number of times sufficient to ensure that if a tautology is accepted, we will find one. We simulate unrolling by repeating instances of the network flow problems. We determine the precise number of times to unroll based on the structure of the strong connections among arithmetic FSAs and the number of arithmetic variables in each arithmetic FSA (see Section 4).

3 Arithmetic Loops

We address arithmetic loops by casting questions about arithmetic automata as questions about network flows. We present the technique by the example in Figure 3. We consider the path taken as the FSA accepts a string to be a flow. Except at the entrance and exit states, each state's in-flow must equal its out-flow. In other words, if on an accepting path, a state is entered three times, then on the same path it must also be exited three times.

In order to capture this intuition, we label the incoming and outgoing transitions at each state where branching or joining occurs. In the example, we label four transitions as W, X, Y, and Z. The labels become the variable names for the *flow variables.* The value of a flow variable equals the number of times the corresponding transition was taken in some accepting path. For the start state, the final state, and each state with branching or joining, we write *flow balance equations.* The label of each transition entering that state appears on one side of the equation and the label of each transition leaving appears on the other. For the start and final states of the arithmetic automaton, we specify a value of "1" entering and leaving respectively.

Paths through the FSA accept expressions of constants and *arithmetic variables.* A tautology is an expression true for all values of the variables, so we universally quantify the arithmetic variables named in the FSA.

Finally, we write *flow-comparison expressions* to link the flow through the FSA to the semantics of the accepted expression. In flow-comparison expressions, flow variables are multiplied by the expressions on their corresponding paths because each trip through a path adds the expression that labels the path to the accepted string. In Figure 3, $\{W, Y, Z \leftarrow 1; X \leftarrow 0\}$ satisfies the expression, and corresponds to the string "`b + c` $\geq$ `b + c`." Additional expressions can prevent most false positives (*e.g.*, by preventing path variables from taking negative values), but we do not discuss them here due to space constraints.

Tarski's theorem [19] 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 soundness result:

Theorem 1. *If we do not discover a tautology then the FSA does not accept a tautology.*

Furthermore, when two or more arithmetic FSAs are linked in a linear structure by logical connectors (*e.g.*, "AND" or "OR"), we can merge in a natural way the equations we generate to model the arithmetic automata, and the soundness result holds for the sequence of automata:

Theorem 2. *If we do not discover a tautology, then the linear chain of arithmetic FSAs does not accept a tautology.*

Incompleteness Allowing the variables to range over real numbers does leave a margin of incompleteness. If the flow variables take on non-integral values, they will not correspond to any path through the FSA.

4 Logical Loops

Consider an arithmetic FSA with an OR-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 "OR" may be a tautology.

Unfortunately, we cannot use equations to address logical loops as we did for arithmetic loops. If we did, the equations for arithmetic loops would not be expressible in first-order arithmetic. Instead, we "unroll" the loop enough times that if the loop accepts some tautology, the unrolling must also accept some tautology. This section presents our technique for discovering tautologies in the presence of logical loops by explaining how we address transitions labeled with each of the three logical keywords: NOT, OR, and AND, and parentheses at the logical level.

4.1 NOT-transitions

The first phase of the analysis takes as input an FSA $\mathcal{A}$ and transforms it into an FSA $\mathcal{A}'$, such that the sets of expressions that $\mathcal{A}$ and $\mathcal{A}'$ accept are logically equivalent, and $\mathcal{A}'$ has no transitions labeled "NOT." Figure 4 illustrates this transformation. Labeled states represent FSAs that accept comparison expressions. Because "AND" has a higher precedence than "OR," applying DeMorgan's law to a negated expression requires that parentheses be added to preserve the precedence in the original expression. However, because we are dealing with FSAs, not single expressions, adding parentheses along one

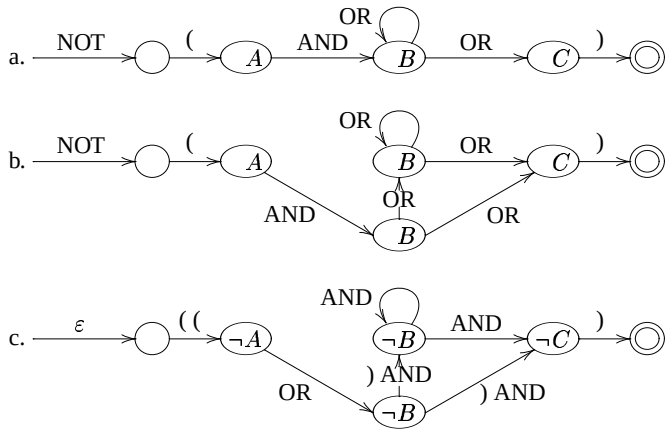


Figure 4. Removing "NOT" from a boolean FSA.

path may lead to imbalanced parentheses on another path. To address this, the transformation first duplicates states that have differently labeled incoming or outgoing transitions. For example, the state B in the original FSA in Figure 4a has incoming transitions labeled "AND" and "OR," so it gets duplicated as in Figure 4b. The transformation then adds parentheses at transitions that terminate sequences of AND-transitions, flips the AND's and the OR's, and flags the states with "¬." When a state is flagged with "¬," the comparison operators in the arithmetic FSA get swapped with their opposites (*e.g.*, $< \rightleftarrows \geq$). Figure 4c shows the last step on the example.

4.2 OR-transitions

By Theorem 2, we can determine whether a linear FSA accepts any tautologies. In this section, given an arbitrary FSA which has only OR-transitions, we generate a finite set of linear FSAs such that at least one accepts a tautology iff the original FSA accepts a tautology.

If all strongly connected components (SCCs) in an FSA are viewed as single states, the FSA is acyclic and all paths through it can be enumerated. The paths can be used to produce a finite set of linear FSAs, and the original FSA accepts a tautology iff one of the linear FSAs accepts a tautology. The following lemma allows us to transform complex looping structures of SCCs into linear sequences of states with self-loops, as in Figure 5.

Lemma 3. *Let $\mathcal{A}$ be an FSA with only OR-transitions which is linear except for SCCs. If $\mathcal{A}$ is transformed into $\mathcal{A}'$ by allowing only unique incoming and outgoing transitions for each state (so that $\mathcal{A}'$ is linear) and adding a self-loop to each state which was originally in an SCC, $\mathcal{A}$ accepts a tautology iff $\mathcal{A}'$ accepts a tautology.*

Lemma 3 follows directly from the commutative property of "OR." If we can determine the maximum number of times each state with a self-loop must be visited to discover a tautology, we can "unroll" the self-loops that number of times to produce linear FSAs. The following theorem yields this number:

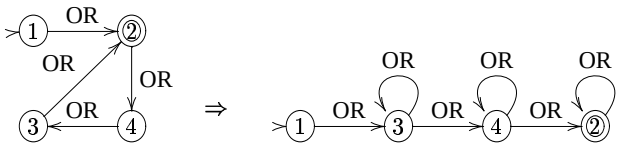


Figure 5. Transforming a complex looping structure into multiple self-loops.

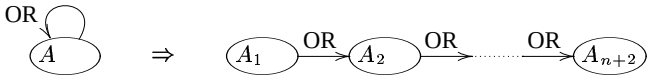


Figure 6. Logical-loop unrolling.

Theorem 4 (Key Theorem). *Let $T = t_1 \vee \ldots \vee t_m$, where each t_i is a comparison of linear arithmetic expressions. Let $\mathfrak{s}$ map expressions of comparisons to sets of the same comparisons, so that $\mathfrak{s}(T) = \{t_1, \ldots, t_m\}$. T is a tautology, iff there exists some tautology T', such that $\mathfrak{s}(T') \subseteq \mathfrak{s}(T)$ and $|\mathfrak{s}(T')| \leq n + 2$, where n is the number of variables named in T.*

We defer the proof of Theorem 4 to Sect. 6. Figure 6 illustrates how we use Theorem 4: if A represents an arithmetic FSA, and a total of n distinct arithmetic variables label the transitions of the FSA, the loop can be unrolled $n + 2$ times to guarantee that if the loop accepts all or part of a tautology, the unrolling does too.

4.3 AND-transitions

This section extends the algorithm from Section 4.2 to deal with AND-transitions. Because "AND" has a higher precedence than "OR," we cannot simply put self-loops on all states in an SCC. The following definitions will be useful in our algorithm:

Definition 5 (AND-chain). *An* AND-chain *is a sequence of states in an SCC connected sequentially by AND-transitions where OR-transitions in the SCC immediately precede and follow the first and last states in the sequence respectively.*

Definition 6 (Minimal AND-chain set). *The* minimal AND-chain set *of an SCC in an FSA is a subset S of the set of all AND-chains of an SCC, such that there are no pairs of AND-chains where the states in one AND-chain form a subset of the states in the other.*

Lemma 7. *Let $\mathcal{A}$ be an FSA with OR- and AND-transitions, and let $\mathcal{A}$ be linear except for SCC's which are entered and exited through OR-transitions. Let $\mathcal{A}$ be transformed into $\mathcal{A}'$ by replacing the SCC's with their minimal AND-chain sets, connecting them linearly with OR-transitions, and adding an OR-transition from the last to the first state of each AND-chain. $\mathcal{A}$ accepts a tautology, iff $\mathcal{A}'$ accepts a tautology.*

Lemma 7 follows first from the commutative property of "OR" because the order in which AND-chains occur does not influence whether or not a tautology is accepted. The minimal AND-chain set can be used because the conjunction of two non-tautologies

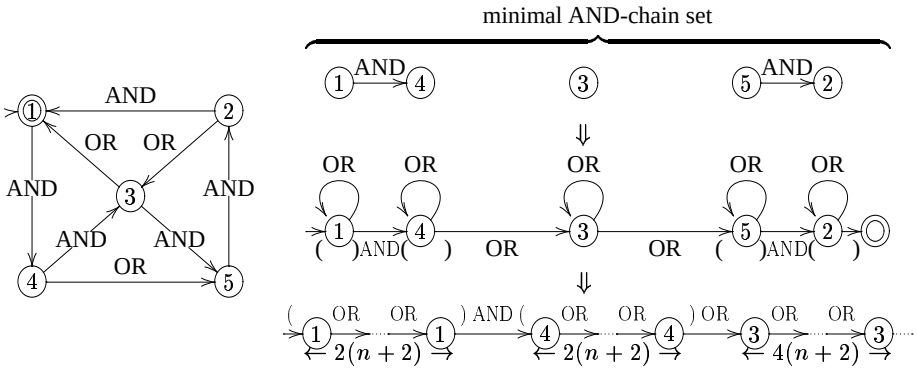


Figure 7. Forming a linear FSA from a strongly connected component to discover tautologies.

can never form a tautology. An algorithm to construct this set finds all states in an SCC with incoming OR-transitions and from those states all acyclic paths which terminate at the first encountered state with an outgoing OR-transition. Figure 7 shows the minimal AND-chain set for an example SCC.

Lemma 7 specifies a transformation from FSA $\mathcal{A}$ to $\mathcal{A}'$ such that $\mathcal{A}$ accepts a tautology iff $\mathcal{A}'$ accepts a tautology. The distributive property of "AND" can be used to transform $\mathcal{A}'$ into a linear FSA of states with self-loops and transitions with parentheses which accepts a tautology iff $\mathcal{A}'$ accepts a tautology. We create such an FSA directly from the AND-chains, as shown in Figure 7.

We can put an upper bound on the number of times each self-loop must be unrolled using Theorem 4. To find this number, we consider an example. Suppose an SCC has two AND-chains: (1) and (2)–(3). From these AND-chains we can construct a linear FSA $\mathcal{A}$ with self-loops as in Figure 7. We can also construct two sets of states where each set has exactly one state from each AND-chain: $\{$(1), (2)$\}$ and $\{$(1), (3)$\}$. From these sets we can construct FSAs $\mathcal{A}_1$ and $\mathcal{A}_2$ where both $\mathcal{A}_1$ and $\mathcal{A}_2$ have only OR-transitions and the states have self-loops. The FSA $\mathcal{A}$ accepts a tautology iff $\mathcal{A}_1$ and $\mathcal{A}_2$ each accepts a tautology. The "only if" direction is straightforward. To prove the "if" direction, consider that if $\mathcal{A}_1$ accepts the tautology "e_1 `OR` e_2," and $\mathcal{A}_2$ accepts the tautology "e_1' `OR` e_3," then $\mathcal{A}$ accepts "e_1 `OR` e_1' `OR (`e_2`) AND (`e_3`)`." This expression in conjunctive normal form is "`(`e_1 `OR` e_1' `OR` e_2`) AND (`e_1 `OR` e_1' `OR` e_3`)`," a tautology. By Theorem 4 the self-loops in $\mathcal{A}_1$ and $\mathcal{A}_2$ need be unrolled at most $n+2$ times, where n is the number of variables that label the transitions in $\mathcal{A}_1$ and $\mathcal{A}_2$. A self-loop over state i in $\mathcal{A}$ must be unrolled $m(n+2)$ times, where m is the product of the numbers of states in the AND-chains which do not include state i. Figure 7 shows the final FSA with the unrollings of self-loops.

4.4 ()-transitions

This section extends the algorithm from Section 4.3 to deal with transitions labeled "(" and ")." Because parentheses have a higher precedence than "AND," we discover AND-chains only among states and transitions of the FSA that have a common parenthetic nesting depth. Recall from Section 2 that parentheses must be balanced on all paths,

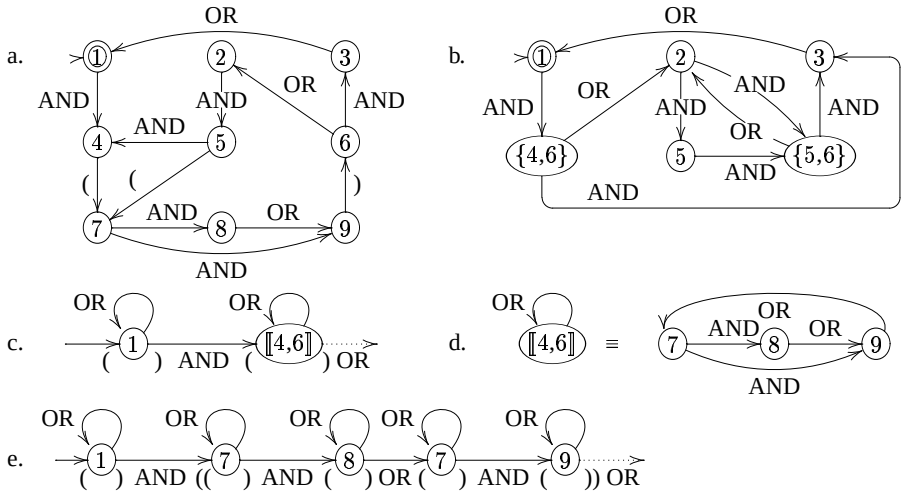


Figure 8. Forming a linear FSA from a strongly connected component with parentheses.

and each state has a unique parenthetic nesting depth. Figure 8 illustrates this algorithm on the FSA in Figure 8a. Before the algorithm discovers AND-chains at depth i, it collapses pairs of states that enter/exit depth $i+1$ into single states, and temporarily removes all states and transitions at depths $> i$. For example, in Figure 8a, states (4) and (5) enter depth 1 and state (6) exits, so $\{4,6\}$ is one pair and $\{5,6\}$ is another pair. Figure 8b shows the FSA with collapsed states ($\{4,6\}$) and ($\{5,6\}$). The *meaning* of a collapsed states ($\{q_s,q_t\}$) is the sub-automaton that can be entered from state q_s and exited from state q_t, and is written ($[\![q_s,q_t]\!]$). The algorithm finds all AND-chains in the FSA, creates a linear FSA with self-loops (as in Figure 7), and replaces collapsed states with their meanings. Figure 8c shows only the beginning of this FSA in order to use the AND-chain (1)–($[\![4,6]\!]$) as an example. Figure 8d shows the sub-automaton that ($[\![4,6]\!]$) with a self-loop represents. In order to "unroll" the self-loop on ($[\![4,6]\!]$), the algorithm recurses on the represented sub-automaton. In this case, the sub-automaton has AND-chains (7)–(8) and (7)–(9). The algorithm produces a linear FSA with self-loops for this sub-automaton, and puts it in place of ($[\![4,6]\!]$). Figure 8e shows the result. When the FSA has no more collapsed states, the self-loops can be unrolled as in Figure 7.

The algorithm for analyzing FSAs is both sound and complete:

Theorem 8 (Soundness and Completeness). *Given a decision procedure for flow-comparison expressions, our algorithm discovers a tautology in an FSA $\mathcal{A}$ iff $\mathcal{A}$ accepts a tautology and accepts only syntactically correct expressions of comparisons of linear arithmetic expressions.*

Theorem 8 follows from Lemma 7 and the distributive property of "AND." A tautology discovered in a linear FSA can be mapped back to a path in the original FSA for the purpose of a useful error message.

5 Complexity

The removal of NOT-transitions (Section 4.1) runs in time linear in the size of $\mathcal{A}$, *i.e.*, $O(|F|)$, and expands $\mathcal{A}$ by a constant factor. The number of paths through $\mathcal{A}$ is exponential in the number of "acyclic" (cannot be reached from themselves) states in $\mathcal{A}$, *i.e.*, $O(2^{|F_{\text{acyc}}|})$. Each path is a query to decision procedure. The number of AND-chains is exponential in the number of "strongly connected" (can be reached from themselves) states in $\mathcal{A}$, *i.e.*, $O(2^{|F_{\text{sc}}|})$. The length of each path is bounded by either the number of acyclic states or the product of the number of AND-chains and the size of the alphabet, *i.e.*, $O(\max(|F_{\text{acyc}}|, 2^{2|F_{\text{sc}}|}|\Sigma|))$. Therefore the number of queries is exponential and the size of each query is also exponential. For this analysis, we consider each query as being sent to an oracle.

6 Proof of Key Theorem

We start with some notational conventions and then give the proof of Theorem 4.

Notational Conventions Let $\mathfrak{s}$ map expressions of comparisons to sets of the same comparisons, so that $\mathfrak{s}(T) = \{t_1, \ldots, t_m\}$. Let T be a tautology and have the form $t_1 \vee t_2 \vee \ldots$, where each t_i is a comparison of two linear arithmetic expressions over n variables. Let $a, b, c, \ldots$ be inequalities in $\Re^n$, let $A, B, C, \ldots$ be sets of inequalities in $\Re^n$, let p, q be points in $\Re^n$, and let $\boldsymbol{a}, \boldsymbol{b}, \ldots$ be vectors normal to $a, b, \ldots$ in $\Re^n$. Let all inequalities a be normalized to $a_1 v_1 + \ldots + a_n v_n \geq 0$ (or "> 0"). Let $A^\intercal$ be a matrix where for each $a \in A$, one column in $A^\intercal$ is $a_1 \ldots a_n$. Let $a(p)$ be the truth value of a at point p (that is, with $p_1, \ldots, p_n$ assigned to a's variables), let $\bigwedge A$ be the conjunction of inequalities in A, and let $\bigwedge A(p)$ be the conjunction of truth values of inequalities in A at point p. Let $a_=$ be the equation formed from a by replacing the comparison operator with '=', and let $A_=$ be defined as $\{a_= \in A_= |\ a \in A\}$. Also, let $d(\cdot, \cdot)$ be an overloaded Cartesian distance function: if its arguments are two points in $\Re^n$, it is the distance between them; and if one argument is a point p and the other an inequality a, it is the distance between p and q where $a(q)$ holds and $d(p, q)$ is minimized.

Proof. [of Theorem 4] Given tautology T, let T' be a tautology such that $\mathfrak{s}(T') \subseteq \mathfrak{s}(T)$, $|\mathfrak{s}(T')| = m$, and for all other tautologies T'', if $\mathfrak{s}(T'') \subseteq \mathfrak{s}(T)$, then $|\mathfrak{s}(T'')| \geq m$. By Lemma 9, if the comparison operators $\{=, \neq\}$ are prohibited, the maximum value for m is decreased by 1. By Lemma 10, if the comparison operators $\{=, \neq\}$ are prohibited, $m \leq n + 1$, where n variables are named in T. Therefore, $m \leq n + 2$. □

Lemma 9. *For all tautologies T, prohibiting the t_i's from using $\{=, \neq\}$ decreases the maximum number of comparison expressions needed for validity by at most 1.*

Proof. Because $e_1 \neq e_2$ is equivalent to $e_1 < e_2 \vee e_1 > e_2$, prohibiting the use of "$\neq$" does not decrease the maximum number of comparison expressions needed for validity.

Suppose for contradiction that there exists some T, $(e_1 = e_2)$, and $(e_3 = e_4)$ such that: the e_i's are linear arithmetic expressions, $T \vee (e_1 = e_2)$ and $T \vee (e_3 = e_4)$ are both not valid, and $T \vee (e_1 = e_2) \vee (e_3 = e_4)$ is valid. By our assumption, there exist points p and r such that $(e_1 = e_2)(p)$ and $(e_3 = e_4)(r)$ hold, but $(e_1 = e_2)(r)$, $(e_3 = e_4)(p)$, $T(p)$, and $T(r)$ do not hold. Because the set of real numbers is dense, there exists point q between p and r. Because $(e_1 = e_2)(r)$ and $(e_3 = e_4)(p)$ do not hold, $(e_1 = e_2)(q)$

and $(e_3 = e_4)(q)$ do not hold. Because $T \vee (e_1 = e_2) \vee (e_3 = e_4)$ is valid, $T(q)$ must hold. For all t_i in T, if $t_i(q)$ holds, $t_i(p)$ or $t_i(r)$ holds, so $T(p)$ or $T(r)$ holds. This contradicts the claims that $T(p)$ and $T(r)$ do not hold. Therefore, prohibiting the use of "=" does not decrease the maximum number of comparison expressions needed for validity by more than 1. □

Lemma 10. *For all tautologies T where "=" and "$\neq$" are not used, there exists some tautology T' such that $\mathfrak{s}(T') \subseteq \mathfrak{s}(T)$ and $|\mathfrak{s}(T')| \leq n + 1$.*

Proof. Overview: Given tautology T, $\neg T$ is an unsatisfiable conjunction. We can take from $\mathfrak{s}(\neg T)$ a smallest pair (A, b) such that $\bigwedge A \Rightarrow \neg b$. The pair (A, b) is a smallest pair such that $\bigwedge A \Rightarrow \neg b$, iff $\bigwedge A \Rightarrow \neg b$ and the vector $\boldsymbol{b}$ normal to b is a negative linear combination of the vectors $\boldsymbol{a_i}$ normal to $a_i \in A$. Because at most n vectors are needed to construct a new vector in $\Re^n$, $|A| \leq n$ and $|A \cup \{b\}| \leq (n + 1)$.

Given tautology T, $\neg T$ is an unsatisfiable conjunction. Let $S = \neg T$. Consider the set P of all pairs (A, b) such that $A \subset \mathfrak{s}(S)$, $b \in \mathfrak{s}(S)$, $b \notin A$, $\bigwedge A$ is satisfiable, and $(\bigwedge A) \wedge b$ is unsatisfiable. Select from P some "smallest" pair (A, b), such that for all other pairs $(A', b') \in P$, $|A'| \geq |A|$.

Vectors only specify direction and magnitude; they do not specify a beginning point, so the implicit beginning is the origin. In order to ensure that an argument about vectors applies to A and b, we will translate the origin such that the hyperplanes defined by A all pass through it. We then create a new inequality c such that the hyperplane defined by c passes through the origin and $\bigwedge A \Rightarrow c$. More specifically, $\exists p.\ \bigwedge A_{=}(p)$, and translating the origin O to p maintains implications. Define c such that $c_{=}(O)$ holds and $c \wedge b$ is unsatisfiable (*i.e.*, $c \Rightarrow \neg b$). If $\exists a_{>} \in A$, then $c = c_{>}$, *i.e.*, c's comparison operator is ">." Otherwise, $c = c_{\geq}$. Therefore, $\bigwedge A \Rightarrow c$.

We now prove that $\boldsymbol{c}$ is a positive linear combination of $A^\intercal$, by showing that if it is not then $\bigwedge A \not\Rightarrow c$, which is false. We show this in three cases. First, by Lemma 11, $\boldsymbol{c}$ must be some linear combination of $A^\intercal$:

$$(\neg \exists \boldsymbol{x}.\ A^\intercal \boldsymbol{x} = \boldsymbol{c}) \ \Rightarrow (\bigwedge A \not\Rightarrow c)\,, \tag{1}$$

Second, by Lemma 12, $\boldsymbol{c}$ must not be a negative linear combination of $A^\intercal$:

$$(\exists \boldsymbol{x}.\ (A^\intercal \boldsymbol{x} = \boldsymbol{c} \ \wedge\ \forall x_i \in \boldsymbol{x}.\ x_i \leq 0)) \ \Rightarrow\ (\bigwedge A \Rightarrow \neg c) \Rightarrow\ (\bigwedge A \not\Rightarrow c) \tag{2}$$

Third, by Lemma 13, $\boldsymbol{c}$ must not be both expressible as a linear combination of $A^\intercal$ and not expressible as either a positive or a negative linear combination of $A^\intercal$:

$$\begin{array}{c}(\exists \boldsymbol{x}.\ A^\intercal \boldsymbol{x} = \boldsymbol{c}) \wedge \\ (\forall \boldsymbol{x}.\ (A^\intercal \boldsymbol{x} = \boldsymbol{c} \ \Rightarrow\ \exists x_i, x_j \in \boldsymbol{x}.\ x_i > 0 \ \wedge\ x_j < 0))\end{array} \Rightarrow\ (\bigwedge A \not\Rightarrow c) \tag{3}$$

Together, (1), (2), and (3) imply:

$$(\bigwedge A \Rightarrow c) \ \Rightarrow\ (\exists \boldsymbol{x}.\ (A^\intercal \boldsymbol{x} = \boldsymbol{c} \ \wedge\ \forall x_i \in \boldsymbol{x}.\ x_i \geq 0)) \tag{4}$$

Finally, we have the converse of (4) by Lemma 12:

$$(\exists \boldsymbol{x}.\ (A^\intercal \boldsymbol{x} = \boldsymbol{c} \ \wedge\ \forall x_i \in \boldsymbol{x}.\ x_i \geq 0)) \ \Rightarrow\ (\bigwedge A \Rightarrow c) \tag{5}$$

If A is not a linearly independent set, then for some $a \in A$ which is a linear combination of $A \backslash \{a\}$, c is a positive linear combination of $A \backslash \{a\}$, and $\bigwedge(A \backslash \{a\}) \Rightarrow c$. This contradicts our assumption that (A, b) is a smallest pair such that $\bigwedge A \Rightarrow \neg b$. Because at most n vectors can be linearly independent in $\Re^n$, $|A| \leq n$ and $|\mathfrak{s}(T')| \leq n + 1$. □

Lemma 11. *Given A and c such that A is satisfiable and $\bigwedge(A_= \cup \{c_=\})(O)$ holds, if $\neg\exists \boldsymbol{x}.\ A^\intercal \boldsymbol{x} = \boldsymbol{c}$, then $\bigwedge A \not\Rightarrow c$.*

Proof. Assume that $\boldsymbol{c}$ is not a linear combination of $A^\intercal$. We will construct a point that demonstrates that $A \not\Rightarrow c$.

Let $\boldsymbol{c} = \boldsymbol{c_A} + \boldsymbol{c_{\neg A}}$, where $\exists \boldsymbol{x}.\ A^\intercal \boldsymbol{x} = \boldsymbol{c_A}$ and $\forall \boldsymbol{a} \in A^\intercal.\ \boldsymbol{a} \cdot \boldsymbol{c_{\neg A}} = 0$. Let p be a point such that $\bigwedge A(p)$ holds. Let vector $\boldsymbol{g} = -\boldsymbol{c_{\neg A}} + \varepsilon(p - O)$, where ε is an arbitrarily small positive number. The point $O + \boldsymbol{g}$ demonstrates that $\bigwedge A \not\Rightarrow c$. Because $\| -\boldsymbol{c_{\neg A}}\| \gg \|\varepsilon(p - O)\|$, $\boldsymbol{c} \cdot \boldsymbol{g} < 0$. Therefore $\boldsymbol{c}(O + \boldsymbol{g})$ does not hold. Because $\forall \boldsymbol{a} \in A^\intercal . \boldsymbol{a} \cdot \boldsymbol{c_{\neg A}} = 0$ and $\varepsilon(p - O) \cdot \boldsymbol{a} > 0$, $\bigwedge A(O + \boldsymbol{g})$ holds. □

Lemma 12. *Given A and c such that A is satisfiable and $\bigwedge(A_= \cup \{c_=\})(O)$ holds, if $\exists \boldsymbol{x}.(A^\intercal \boldsymbol{x} = \boldsymbol{c}) \wedge (\forall x_i \in \boldsymbol{x}. x_i \geq 0)$ (*i.e.*, $\boldsymbol{c}$ is a positive linear combination of $A^\intercal$), then $\bigwedge A \Rightarrow c$.*

Proof. Consider a point p such that $\forall a_i \in A. a_i(p)$. We can expand $a_i(p)$ to $a_{i1}p_1 + \ldots + a_{in}p_n \geq 0$ (or "> 0"). Let λ_i be the left-hand side of the expression. Because $a_i(p)$ holds, $\lambda_i \geq 0$. By our assumption, $x_1, \ldots, x_n \geq 0$. Use the following inference rule: $x_1, \ldots, x_n, \lambda_1, \ldots, \lambda_n \geq 0 \ \Rightarrow\ \lambda_1 x_1 + \cdots + \lambda_n x_n \geq 0$. By definition, the right-hand side of the inference rule equals $c(p)$. Because point p was picked arbitrarily, this holds for all points, and $\bigwedge A \Rightarrow c$. □

Lemma 13. *Given A and c such that A is satisfiable and $\bigwedge(A_= \cup \{c_=\})(O)$ holds, if for all $\boldsymbol{x}$ such that $A^\intercal \boldsymbol{x} = \boldsymbol{c}$, $\exists x_i, x_j \in \boldsymbol{x}.\ x_i, -x_j > 0$, then $\bigwedge A \not\Rightarrow c$.*

Proof. This proof requires that $A^\intercal$ have a non-zero determinant. If $A^\intercal$ does not span the vector space of $\Re^n$, we must augment A. Let $A^\perp$ be a basis for the orthogonal complement of the vector space spanned by $A^\intercal$. Let $|A^\perp| = m$. Let $A^{\perp'}$ be the set of inequalities created from the vectors in $A^\perp$. The choice of $A^\perp$ is arbitrary within the constraint that it be a basis for $A^\intercal$'s orthogonal complement, and it does not affect the following argument. For the rest of this proof, we consider A to be augmented.

By Cramer's Rule [8], we know that for

$$\begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}, \quad x_i = \frac{\det(A_i)}{\det(A^\intercal)}$$

where

$$A_i = \begin{bmatrix} a_{11} & \cdots & a_{(i-1)1} & c_1 & a_{(i+1)i} & \cdots & a_{n1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{1n} & \cdots & a_{(i-1)n} & c_n & a_{(i+1)n} & \cdots & a_{nn} \end{bmatrix}.$$

Because $A^\intercal$ spans $\Re^n$, $\det(A^\intercal) \neq 0$. We will use this definition to find a point in $\Re$ that demonstrates $A \not\Rightarrow c$.

Let $\boldsymbol{y} = \det(A^\intercal) \cdot \boldsymbol{x}$. By the assumption of the lemma, $\exists x_i, x_j \in \boldsymbol{x}.(x_i > 0) \wedge (x_j < 0)$, so $\exists y_i, y_j \in \boldsymbol{y}.(y_i > 0) \wedge (y_j < 0)$. By definition,

$$\det(A^\intercal) = \sum(\pm) a_{1\ell_1} a_{2\ell_2} \cdots a_{n\ell_n}$$

where the summation ranges over all permutations $\ell_1 \ell_2 \cdots \ell_n$ of the set $\{1, 2, \ldots, n\}$. The sign is $+$ or $-$ according to whether the permutation $\ell_1 \ell_2 \cdots \ell_n$ is even or odd.

We wish to find some vector $\boldsymbol{v}$ such that for all $a \in A$, $\boldsymbol{a} \cdot \boldsymbol{v} > 0$, but $\boldsymbol{v} \cdot \boldsymbol{c} = 0$. We will first find a vector $\boldsymbol{e}$ such that $\boldsymbol{c} \cdot \boldsymbol{e} = 0$, for some $a_i, a_j \in A$, $\boldsymbol{a_i} \cdot \boldsymbol{e} > 0$ and $\boldsymbol{a_j} \cdot \boldsymbol{e} > 0$, and for all other $a \in A$, $\boldsymbol{a} \cdot \boldsymbol{e} = 0$. Let $y_i > 0$ and $y_j < 0$. We can "factor out" $\boldsymbol{a_j}$ from y_i as $y_i = \det(A_i) =$

$$\sum_{k=1}^{n} a_{jk} \left(\sum(\pm) a_{1\ell_i} \cdots a_{(j-1)\ell_{(j-1)}} a_{(j+1)\ell_{(j+1)}} \cdots a_{(i-1)\ell_{(i-1)}} c_{\ell_i} a_{(i+1)\ell_{(i+1)}} \cdots a_{n\ell_n}\right)$$

From what remains, we form the vector $\boldsymbol{e}$:

$$e_j = \sum(\pm) a_{1\ell_i} \cdots a_{(j-1)\ell_{(j-1)}} a_{(j+1)\ell_{(j+1)}} \cdots a_{(i-1)\ell_{(i-1)}} c_{\ell_i} a_{(i+1)\ell_{(i+1)}} \cdots a_{n\ell_n}$$

So, $\boldsymbol{e} \cdot \boldsymbol{a_j} = \det(A_i) = y_i > 0$. Factoring out $\boldsymbol{a_j}$ from $\det(A_i)$ to get $\boldsymbol{e}$ and multiplying $\boldsymbol{e} \cdot \boldsymbol{a_i}$ has the effect of replacing $\boldsymbol{a_j}$ by $\boldsymbol{a_i}$ in A_i to get A_i' and finding $\det(A_i')$. This new matrix, A_i', looks like

$$\begin{bmatrix} a_{11} & \cdots & a_{(j-1)1} & a_{i1} & a_{(j+1)1} & \cdots & a_{(i-1)1} & c_1 & a_{(i+1)i} & \cdots & a_{n1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{1n} & \cdots & a_{(j-1)n} & a_{in} & a_{(j+1)n} & \cdots & a_{(i-1)n} & c_n & a_{(i+1)n} & \cdots & a_{nn} \end{bmatrix}$$

By swapping $\boldsymbol{a_i}$ and $\boldsymbol{c}$ in A_i', we get

$$\begin{bmatrix} a_{11} & \cdots & a_{(j-1)1} & c_1 & a_{(j+1)1} & \cdots & a_{(i-1)1} & a_{i1} & a_{(i+1)i} & \cdots & a_{n1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{1n} & \cdots & a_{(j-1)n} & c_n & a_{(j+1)n} & \cdots & a_{(i-1)n} & a_{in} & a_{(i+1)n} & \cdots & a_{nn} \end{bmatrix}$$

which is exactly A_j. It is a property of determinants that if matrix B results from matrix A by interchanging two rows/columns of A, then $\det(B) = -\det(A)$. Therefore, $\boldsymbol{e} \cdot \boldsymbol{a_i} = \det(A_i') = -\det(A_j) = -y_j$. Consequently, $\boldsymbol{e} \cdot \boldsymbol{a_i} > 0$ iff $y_j < 0$, which, by our assumption, it is.

Now suppose we find $\boldsymbol{e} \cdot \boldsymbol{c}$. This is equivalent to finding $\det(A_i'')$, where $A_i'' =$

$$\begin{bmatrix} a_{11} & \cdots & a_{(j-1)1} & c_1 & a_{(j+1)1} & \cdots & a_{(i-1)1} & c_1 & a_{(i+1)i} & \cdots & a_{n1} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{1n} & \cdots & a_{(j-1)n} & c_n & a_{(j+1)n} & \cdots & a_{(i-1)n} & c_n & a_{(i+1)n} & \cdots & a_{nn} \end{bmatrix}$$

It is also a property of determinants that if two rows/columns of A are equal, then $\det(A) = 0$, so $\boldsymbol{e} \cdot \boldsymbol{c} = 0$. This is also the case with $\boldsymbol{e} \cdot \boldsymbol{a_k}$ for $i \neq k \neq j$.

We will now find the vector $\boldsymbol{v}$ we were originally searching for. Let the vector name $\boldsymbol{e_{k,l}}$ denote the property that $\boldsymbol{e_{k,l}} \cdot \boldsymbol{a_k}$, $\boldsymbol{e_{k,l}} \cdot \boldsymbol{a_l} > 0$, $\boldsymbol{e_{k,l}} \cdot \boldsymbol{c} = 0$, and for $k \neq i \neq l$ $\boldsymbol{a_{k,l}} \cdot \boldsymbol{a_i} = 0$. In the previous paragraph, we constructed $\boldsymbol{e_{i,j}}$ given that $(x_i > 0)$ and $(x_j < 0)$. Using the same technique, for all x_k we construct $\boldsymbol{e_{k,j}}$ if $x_k > 0$, and $\boldsymbol{e_{i,k}}$ if $x_k < 0$. If $x_k = 0$, $\boldsymbol{e_{k,k}}$ be $\boldsymbol{a_k}$, which is orthogonal to $\boldsymbol{c}$. Let the set E contain all of these $\boldsymbol{e_{k,l}}$ vectors. Dot products have the property: $\boldsymbol{a} \cdot \boldsymbol{c_1} + \boldsymbol{a} \cdot \boldsymbol{c_2} = \boldsymbol{a} \cdot (\boldsymbol{c_1} + \boldsymbol{c_2})$. If $\boldsymbol{v} = \sum E$, then $\boldsymbol{v} \cdot \boldsymbol{c} = 0$ and for all $a \in A$, $\boldsymbol{v} \cdot \boldsymbol{a} > 0$.

The vectors $\boldsymbol{c}$ and $\boldsymbol{v}$ plus the point O define a plane, and the angle between $\boldsymbol{c}$ and $\boldsymbol{v}$ is $90\,°$. The vector $\boldsymbol{v_{\neg c}} = \boldsymbol{v} + \varepsilon(\boldsymbol{v} - \boldsymbol{c})$ lies in the same plane, and the angle between $\boldsymbol{c}$ and $\boldsymbol{v_{\neg c}}$ is $(90 + \varepsilon')\,°$. Because the set of real numbers is dense, for small ε, we still have that for all $a \in A$, $\boldsymbol{a} \cdot \boldsymbol{v_{\neg c}} > 0$, so $\bigwedge A(O + \boldsymbol{v_{\neg c}})$ holds. Because $\boldsymbol{c} \cdot \boldsymbol{v_{\neg c}} < 0$, $c(O + \boldsymbol{v_{\neg c}})$ does not hold. Thus $A \not\Rightarrow c$. □

7 Related Work

In this section, we survey 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 [19]. Collins used cylindrical decomposition to check validity in the same theory more efficiently, but his algorithm also has high complexity [7]. The first-order theory over integers is undecidable because of the undecidability of solving Diophantine equations [9]. However, an important fragment, Presburger Arithmetic, is decidable.

Linear Constraints In program analysis and formal verification, decision procedures for linear constraints are widely used. Some proposed techniques include Fourier-Motzkin variable elimination [15], the Sup-Inf method of Bledsoe [5], and Nelson's method based on Simplex [11]. 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 [14]. Shostak considers a slightly more general problem $ax + by \leq k$, where a, b, and k are integer constants [16]. 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 [1]. Su and Wagner leverage ideas from Pratt and Shostak to propose the first polynomial time algorithm for a general class of integer range constraints [18].

Decision Procedures based on Combined Theories In 1979, Nelson and Oppen proposed a method for combining theories in a decision procedure [12]. Contemporary theorem provers, such as in Necula and Lee's certifying compiler [10], 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 [17]. 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 [13], STeP [4], and SVC [3]. 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 [2] is a descendant of SVC that includes a builtin SAT solver and support for quantifiers.

8 Conclusions and Future Work

We have introduced the validity checking problem for finite automata over linear arithmetic constraints, motivated by the need for advanced checking of meta-programs. More importantly, we have presented the first decision procedure for this validity problem, which enables the implementation of formal analysis tools for meta-programs. Our decision procedure is based on a novel use of network-flow problems and a theorem bounding the number of constraints needed in a disjunctive tautology. For future work, we plan to design and implement a tool to check for or verify the absence of database command injection problems in web and database applications. It might also be interesting to consider the dual problem, namely the satisfiability problem. Our proof technique for validity seems also applicable for satisfiability. Finally, it is interesting to investigate the application of our decision procedure in other settings of analyzing meta-programs.

References

1. B. Aspvall and Y. Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. *SIAM Computing*, 9(4):827–845, 1980.
2. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. CAV'04*, July 2004.
3. C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity Checking for Combinations of Theories with Equality. In *Proc. FMCAD'96*, pages 187–201, 1996.
4. N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. CAV'96*, pages 415–418, 1996.
5. W. Bledsoe. The Sup-Inf method in Presburger arithmetic. Technical report, University of Texas Math Department, Dec. 1974.
6. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS'03*, pages 1–18, 2003. URL: `http://www.brics.dk/JSA/`.
7. G. E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, 1975.
8. B. Kolman. *Introductory Linear Algebra with Applications*. Prentice Hall, 1997.
9. Y. Matiyasevich. Solution of the tenth problem of hilbert. *Mat. Lapok*, 21:83–87, 1970.
10. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. PLDI'98*, pages 333–344, 1998.
11. G. Nelson. Techniques for program verification. Technical report, Xerox PARC, 1981.
12. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.
13. S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. In *CADE 11*, 1992.
14. V. Pratt. Two easy theories whose combination is hard. Technical report, MIT, Sept. 1977.
15. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. Supercomputing*, pages 4–13, 1991.
16. R. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4), 1981.
17. R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.
18. Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *Proc. TACAS'04*, 2004.
19. A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.