Verification of Data Paths Using Unbounded Integers: Automata Strike Back

Tobias Schuele and Klaus Schneider

Reactive Systems Group, Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{Tobias.Schuele,Klaus.Schneider}@informatik.uni-kl.de
http://rsg.informatik.uni-kl.de

Abstract. We present a decision procedure for quantifier-free Presburger arithmetic that is based on a polynomial time translation of Presburger formulas to alternating finite automata (AFAs). Moreover, our approach leverages the advances in SAT solving by reducing the emptiness problem of AFAs to satisfiability problems of propositional logic. In order to obtain a complete decision procedure, we use an inductive style of reasoning as originally proposed for proving safety properties in bounded model checking. Besides linear arithmetic constraints, our decision procedure can deal with bitvector operations that frequently occur in hardware design. Thus, it is well-suited for the verification of data paths at a high level of abstraction.

1 Introduction

Hardware verification is usually performed at the level of propositional logic which is self-evident if the system to be verified is given as a netlist of gates. Using propositional logic as the basic formalism allows one to perform a symbolic state space exploration of the system by means of binary decision diagrams (BDDs), or to apply bounded model checking procedures that make use of sophisticated SAT solvers. However, while both approaches have been successfully used for the verification of control-flow intensive systems, large data paths are still hardly tractable using most symbolic model checkers. To solve this problem, various approaches have been proposed such as abstract interpretation, symmetry reduction, partial order reduction, and many others that aim at fighting the state explosion problem.

Another approach to verify data-flow intensive systems is the use of more powerful base logics. This is particularly interesting if the system is given at a higher level of abstraction than gate level, where more complex operations are available. Regarding the verification of data paths, such a logic should at least contain operations for integer arithmetic. However, due to the undecidability of full arithmetic, this either requires the use of interactive theorem provers or to consider decidable fragments such as Presburger arithmetic [1]. In recent years, decision procedures for such decidable logics have attained increasing interest, not only as stand-alone procedures, but also as the basis for combined decision procedures. For instance, the UCLID system [2] that is based on a combination of the theory of uninterpreted functions with equality, Presburger arithmetic, and the theory of arrays has been successfully used for the verification of complex micro-processors.

E. Bin, A. Ziv, and S. Ur (Eds.): HVC 2006, LNCS 4383, pp. 65–80, 2007.

In this paper, we present yet another procedure for checking satisfiability of quantifier-free Presburger arithmetic formulas. Our method is based on the translation of Presburger formulas to finite automata as originally proposed in [3] and enhanced for example in [4,5,6]. It is sometimes argued that automata-based decision procedures for Presburger arithmetic are in general less efficient than other approaches such as integer linear programming and Fourier-Motzkin variable elimination [7]. Indeed, none of the benchmarks used in [8] could be solved using the LASH tool [9] that is based on the method presented in [5]. However, such comparisons certainly depend on the type of automata formulas are translated to and on the underlying decision procedures for propositional logic (BDDs vs. SAT solvers).

In contrast to previous approaches, our method employs alternating finite automata (AFAs) which can be viewed as a generalization of nondeterministic automata. As a major advantage of AFAs, our method can benefit from sophisticated SAT solvers that are state-of-the-art in many areas. In particular, the equational structure of AFAs allows us to unwind their transition relations efficiently which is useful for checking emptiness (a formula is unsatisfiable iff the language of the corresponding automaton is empty). However, simply unwinding an AFA only yields a semi-decision procedure that can be used to prove satisfiability of a formula, but not to prove its unsatisfiability. To solve this problem, we use an inductive style of reasoning that has been originally proposed for checking safety properties in bounded model checking.

As another advantage, our approach can be easily extended to deal with more powerful logics. Regarding the verification of data paths, we consider an extension of Presburger arithmetic by bitvector operations, since these operations frequently occur in hardware design. For example, the ALUs of most microprocessors support arithmetic as well as bitwise operations. While such an extension is straightforward in practice, it has considerable impact on the complexity of the decision procedures: We show that the satisfiability problem of Presburger arithmetic with bitvector operations cannot be reduced to a polynomial sized satisfiability problem of propositional logic. For this reason, we use an inductive approach and do not rely on a polynomial upper bound on the size of the constructed formulas as in [8]. Finally, it should be mentioned that automata encode all solutions of a formula which makes it easy to find the smallest one.

There has been much work on decision procedures for Presburger arithmetic. A comparison of different approaches can be found in [10,11,7]. The construction of deterministic finite automata (DFAs) from linear arithmetic constraints is described in detail in [5]. However, the proposed algorithms perform an explicit enumeration of the state space. A symbolic encoding using BDDs is presented in [7,12]. From a practical point of view, our method is most closely related to the approach presented in [8] that also makes use of SAT solvers. The idea is to reduce the infinite domain of Presburger formulas to a finite one by computing bounds on the size of the solutions. In contrast to our approach, however, it cannot directly deal with bitvector operations as described above. Strichman [13] presents another SAT based decision procedure that is based on Fourier-Motzkin elimination. In the worst case, this approach leads to a SAT problem that is doubly exponential in the size of the formula. Finally, Kroening et. al [14] propose an abstraction-based procedure that combines a SAT solver with a theorem prover in order to successively generate approximations of the original formula.

The outline of the paper is as follows: after briefly describing AFAs and Presburger arithmetic in the next section, we present the corresponding translation in Section 3. Then, we describe our approach for checking emptiness of AFAs and discuss the effect of introducing bitvector operations (Section 4). Experimental results are given in Section 5, and finally, we conclude with a summary and directions for future work.

2 Foundations

2.1 Alternating Finite Automata

Alternating finite automata (AFAs) [15,16,17,18] and also Boolean automata [19,20,21] are a natural generalization of nondeterministic finite automata (NFAs) in the sense that the next state is not just chosen from a set of states, but determined by a propositional formula[1]. Recall that an NFA is a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to \mathscr{P}(Q)$ is the transition function, q_0 is the initial state, and $F \subseteq Q$ is the set of final states. A word $aw \in \Sigma^+$ is accepted in a state $q \in Q$ iff there exists at least one successor state $q' \in \delta(q, a)$ such that w is accepted in q' (the empty word is accepted in q iff $q \in F$). More formally, we recursively define $\mathsf{acc}(q, aw) :\Leftrightarrow \exists q' \in \delta(q, a).\mathsf{acc}(q', w)$. Since there are only finitely many states, the existential[2] acceptance condition of NFAs can be replaced by a disjunction, i.e., $\mathsf{acc}(q, aw) :\Leftrightarrow \bigvee_{q' \in \delta(q,a)} \mathsf{acc}(q', w)$. AFAs extend this idea to allow arbitrary propositional formulas in place of the disjunctions found in NFAs: Instead of a set of successor states, each state has an associated formula that characterizes its acceptance condition. Thus, to decide whether a word is accepted in a state, one simply evaluates the associated formula.

An AFA can be formally defined as follows, where we use Boolean variables not only to represent the states, but also to encode the alphabet, i.e., we assume that a letter is a vector of Boolean values:

Definition 1 (Alternating Finite Automaton (AFA)). *An alternating finite automaton is a tuple (Q, V, δ, I, F), where*

- *Q is the set of state variables,*
- *V is the set of input variables,*
- *$\delta : Q \to \mathsf{Prop}(Q \cup V)$ is the transition function that associates with each state variable a propositional formula over the variables $Q \cup V$,*
- *$I \in \mathsf{Prop}(Q \cup V)$ is the initial formula over the variables $Q \cup V$, and*
- *$F : Q \to \mathbb{B}$ is the final function that maps state variables to the Booleans.*

In the sequel, we denote state variables by $q_0, \ldots, q_m$ and input variables by $v_0, \ldots, v_n$. Moreover, we abbreviate $\Sigma := \mathbb{B}^{|V|}$.

[1] Originally, the term alternation stems from the fact that existential and universal quantifiers can alternate during the course of a computation, whereas in a nondeterministic computation there are only existential quantifiers.

[2] Clearly, one can also define a dual type of automata, where a word is accepted in a state iff all of its successor states accept the remaining word. The acceptance condition of such universal automata is a conjunction of the acceptance conditions over the successor states.

This definition slightly differs from the ones found in the literature in that it provides an initial formula instead of only a single initial state. As a result, an AFA as defined above cannot accept the empty word. Moreover, we allow arbitrary propositional formulas, i.e., a variable may occur not only in positive, but also in negative form.

Definition 2 (Acceptance and Language of an AFA). *Given a propositional formula f, let $f[v_i/g_i, w_j/h_j]_{0\leq j\leq n}^{0\leq i\leq m}$ denote the formula obtained by simultaneously substituting the formulas g_i and h_j for the variables v_i and w_j, respectively, for $0 \leq i \leq m$ and $0 \leq j \leq n$. Given an AFA $\mathcal{A} = (Q, V, \delta, I, F)$, the acceptance of a word with respect to a formula f is defined as follows, where $(b_0, \ldots, b_n) \in \Sigma$ and $w \in \Sigma^+$:*

$$\mathsf{acc}(f, (b_0, \ldots, b_n)\, w) \;:\Leftrightarrow\; \mathsf{acc}(f[q_i/\delta(q_i), v_j/b_j]_{0\leq j\leq n}^{0\leq i\leq m}, w)$$
$$\mathsf{acc}(f, (b_0, \ldots, b_0)) \;:\Leftrightarrow\; f[q_i/F(q_i), v_j/b_j]_{0\leq j\leq n}^{0\leq i\leq m}$$

A word w is accepted by $\mathcal{A}$ iff $\mathsf{acc}(I, w)$ holds. The language accepted by $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) := \{w \in \Sigma^* \mid \mathsf{acc}(I, w)\}$.

AFAs have the property that they are backward deterministic which means that they are deterministic if one considers them working on the input string from right to left. Thus, an AFA can also be viewed as a symbolic description of a deterministic finite automaton (DFA) accepting the reverse language. The transition relation is thereby given as an equation system, i.e., by the conjunction $\bigwedge_{q\in Q} q' \leftrightarrow \delta(q)$, where q' is the next state variable associated with q. For this reason, it is often more convenient to consider the reverse language when dealing with AFAs, which has lead to the notion of reversed AFAs [22,23,24]. In particular, when constructing AFAs for Presburger formulas, we will assume that the input is being read from right to left.

However, it should be emphasized that the order in which the input is read is mainly a matter of taste. For instance, checking whether a word is accepted by an AFA can be done in both directions with essentially the same complexity. The only difference is that when reading from right to left, we have to deal with a vector of formulas, whereas in the opposite direction it suffices to consider a single formula (cf. Definition 2). The crucial point is that AFAs have an equational structure [25], or in terms of symbolic model checking, an explicit partitioning of the transition relation. This allows us to unwind AFAs without introducing additional state variables and to employ efficient SAT solvers for checking emptiness, as mentioned in the introduction.

Given two AFAs $\mathcal{A}_1 = (Q_1, V, \delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, V, \delta_2, I_2, F_2)$, the Boolean operations are defined by the corresponding operations on the initial formulas:

- $\neg\mathcal{A}_1 := (Q_1, V, \delta_1, \neg I_1, F_1)$
- $\mathcal{A}_1 \wedge \mathcal{A}_2 := (Q_1 \cup Q_2, V, \delta_1 \cup \delta_2, I_1 \wedge I_2, F_1 \cup F_2)$
- $\mathcal{A}_1 \vee \mathcal{A}_2 := (Q_1 \cup Q_2, V, \delta_1 \cup \delta_2, I_1 \vee I_2, F_1 \cup F_2)$

It is easy to see that the Boolean operations satisfy the following equations (ϵ denotes the empty word):

- $\mathcal{L}(\neg\mathcal{A}_1) = \overline{\mathcal{L}(\mathcal{A}_1)} \setminus \{\epsilon\}$
- $\mathcal{L}(\mathcal{A}_1 \wedge \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$
- $\mathcal{L}(\mathcal{A}_1 \vee \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$

2.2 Quantifier-Free Presburger Arithmetic with Bitvector Operations

In this subsection, we briefly describe the syntax and semantics of quantifier-free Presburger arithmetic with bitvector operations (QFPAbit).

Definition 3 (Syntax of QFPAbit). *Let $\mathcal{V} := \mathcal{V}_\mathbb{Z} \cup \mathcal{V}_\mathbb{B}$ be a finite set of integer and Boolean variables, respectively, such that $\mathcal{V}_\mathbb{Z} \cap \mathcal{V}_\mathbb{B} = \emptyset$ holds. Then, the set of terms is defined as follows with $c \in \mathbb{Z}$ and $x \in \mathcal{V}_\mathbb{Z}$:*

$$T := c \mid x \mid T + T \mid c \cdot T \mid \overrightarrow{\neg} T \mid T \overrightarrow{\wedge} T \mid T \overrightarrow{\vee} T$$

The set of formulas is defined as follows with $p \in \mathcal{V}_\mathbb{B}$ and $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$:

$$F := p \mid T \bowtie T \mid \neg F \mid F \wedge F \mid F \vee F$$

The semantics should be clear from the context except for the bitvector operations $\overrightarrow{\neg}$, $\overrightarrow{\wedge}$, and $\overrightarrow{\vee}$ that need some further explanation. Their semantics is based on two's complement encoding, where the value $\langle x_k \dots x_0 \rangle_\mathbb{Z}$ of a bitvector $(x_k \dots x_0)$ is defined as follows:

$$\langle x_k \dots x_0 \rangle_\mathbb{Z} := -2^k x_k + \sum_{i=0}^{k-1} 2^i x_i$$

Recall that in this encoding the most significant bit can be replicated without changing the value (sign extension). Thus, the equation $\langle x_k \dots x_0 \rangle_\mathbb{Z} = \langle x_k x_k \dots x_0 \rangle_\mathbb{Z}$ is valid for all bitvectors $(x_k \dots x_0)$. A term $\overrightarrow{\neg} x$ is then interpreted as $\langle \neg x_k \dots \neg x_0 \rangle_\mathbb{Z}$, provided that $x = \langle x_k \dots x_0 \rangle_\mathbb{Z}$ holds. Similarly, the terms $x \overrightarrow{\wedge} y$ and $x \overrightarrow{\vee} y$ are interpreted as $\langle x_k \wedge y_k \dots x_0 \wedge y_0 \rangle_\mathbb{Z}$ and $\langle x_k \vee y_k \dots x_0 \vee y_0 \rangle_\mathbb{Z}$, respectively. For example, as -6 is represented by the bitvector (1010), and 5 by (0101), it follows that $-6 \overrightarrow{\vee} 5$ is represented by (1111), which is the number -1. Moreover, we have $\overrightarrow{\neg} 5 = -6$.

It is well-known that a set can be defined in pure Presburger arithmetic, i.e., without extensions such as bitvector operations, iff it is ultimately periodic [26,27]. A set $Z \subseteq \mathbb{Z}$ is ultimately periodic iff there exists a $p \geq 1$ (the period) such that the following holds:

- $\exists n^+ \geq 0. \forall n \geq n^+. n \in Z \Leftrightarrow n + p \in Z$
- $\exists n^- \leq 0. \forall n \leq n^-. n \in Z \Leftrightarrow n - p \in Z$

However, this does not hold for Presburger arithmetic with bitvector operations as defined above. Consider, for example, the formula

$$\mathsf{pow2}(x) := 1 + ((x-1) \overrightarrow{\vee} x) = 2x \wedge x > 0$$

which holds iff x is a power of two. Since the set of satisfying assignments of pow2 is not ultimately periodic, QFPAbit is strictly more expressive than pure Presburger arithmetic. In fact, QFPAbit is as expressive as the quantifier-free fragment of the weak monadic second order logic of linear order ($\mathsf{WMSO}_<$). The proof is based on the fact that for every $\mathsf{WMSO}_<$ formula there exists an equivalent one whose atoms express singletons, set inclusion, and the successor function. Since these atoms are definable in QFPAbit, every $\mathsf{WMSO}_<$ formula can be translated to an equivalent QFPAbit formula.

3 Translation of Quantifier-Free Presburger Arithmetic to AFAs

The relationship between QFPAbit formulas and AFAs is established via the two's complement encoding presented in the previous subsection. For that purpose, we associate with each integer variable $x \in \mathcal{V}_{\mathbb{Z}}$ and each Boolean variable $p \in \mathcal{V}_{\mathbb{B}}$ exactly one input variable $v \in V$ of an AFA. A word $w \in \Sigma^+$ with

$$w = \begin{pmatrix} b_{0,k} \\ \vdots \\ b_{n,k} \end{pmatrix} \dots \begin{pmatrix} b_{0,1} \\ \vdots \\ b_{n,1} \end{pmatrix} \begin{pmatrix} b_{0,0} \\ \vdots \\ b_{n,0} \end{pmatrix}$$

is then interpreted as the assignment $\xi_w : \mathcal{V} \to \mathbb{Z} \cup \mathbb{B}$, where $\xi_w(x_i) := \langle b_{i,k} \dots b_{i,0} \rangle_{\mathbb{Z}}$ for $x_i \in \mathcal{V}_{\mathbb{Z}}$ and $\xi_w(p_i) := b_{i,k}$ for $p_i \in \mathcal{V}_{\mathbb{B}}$. Using this encoding scheme, the i-th row encodes the value of the i-th variable, and the j-th column is read by an automaton in the j-th step. Hence, the number of variables is finite and fixed, whereas their bitwidth is also finite, but arbitrarily large.

Since we have already shown how to perform the Boolean operations on AFAs, it remains to describe the construction of automata for relations and Boolean variables. Since the latter only depend on the most significant bits of a word, they can be easily translated to an AFA by considering only the initial formula, i.e., a formula p with $p \in \mathcal{V}_{\mathbb{B}}$ is translated to the automaton $(\emptyset, V, \emptyset, v_p, \emptyset)$ with $v_p \in V$ (the third and the fifth component are the empty set, since the domain of the corresponding functions is empty). Hence, there is no overhead for translating the propositional part of a QFPAbit formula to an AFA.

The translation of arbitrary relations is slightly more difficult. As a first step, we separate the bitvector parts from the arithmetic parts. By introducing new variables, it is straightforward to construct an equisatisfiable formula that only contains relations of either type. In the same way, relations over more than three variables can be reduced to relations over at most three variables. For instance, the formula $(x \overrightarrow{\wedge} y) + z = s$ is satisfiable iff the formula $(x \overrightarrow{\wedge} y = t) \wedge (t + z = s)$ is satisfiable. Thus, it suffices to consider the following three types of relations: $\overrightarrow{\neg} x = y$, $x \overrightarrow{\wedge} y = z$, and $x + y \leq z$. In practice, however, this is rather inefficient, since it often requires a large number of auxiliary variables. Moreover, equations must be expressed by a conjunction of inequalities. For this reason, we consider relations of the following types, where T_1 and T_2 are terms containing only bitvector operations:

$$\text{(A) } T_1 = T_2 \qquad \text{(B) } \sum_{i=0}^{n} c_i x_i = c \qquad \text{(C) } \sum_{i=0}^{n} c_i x_i < c$$

Equations of type (A) can be translated to an AFA with a single state variable. Initially, this variable is set to true and at each step it is checked whether the inputs satisfy the equation. If the equation is not satisfied, the state variable is set to false and keeps this value until the last letter has been read. More precisely, let T_1' and T_2' be the terms obtained by replacing all integer variables x_i with the corresponding input variables v_i. Then, the equation $T_1 = T_2$ is translated to the AFA $(\{q\}, V, \delta, I, F)$ with $\delta(q) \equiv I \equiv q \wedge (T_1' \leftrightarrow T_2')$ and $F(q) \equiv 1$ ($\equiv$ denotes equivalence of propositional formulas).

The construction of DFAs from linear arithmetic constraints over natural numbers has already been presented in [4] and extended in [5] to deal with integers. The idea is to read the input from left to right, i.e., starting with the most significant bits, and to keep track of the value of the left-hand side of an equation (inequality) as successive bits are read. Thus, each state corresponds to an integer γ that represents the current value. The next state is then defined by $\gamma' = 2\gamma + \sum_{i=0}^{n} c_i b_i$, where $(b_n, \ldots, b_0)$ is the input vector. For an equation, the final state is uniquely determined by its right-hand side, i.e., the constant c. Similarly, for an inequality, a state is final iff its value is less than c. Moreover, it was shown in [5] that there always exists an $\alpha \in \mathbb{N}$ such that $|\gamma| > \alpha$ implies $|\gamma'| > |\gamma|$. Thus, all states with $|\gamma| > |c|$ can be collapsed into a single nonaccepting state. As a result, there are only finitely many states. More precisely, the number of states is bounded by $O(\log_2 |c| \cdot \sum_{i=0}^{n} |c_i|)$ [5].

In contrast to [5], our approach is based on reading the input from right to left when considering DFAs (the corresponding AFAs still read from left to right, and thus, the most significant bits that determine the signs can be easily encoded in the initial formulas). In many cases, this allows us to detect conflicts very early. For example, given an equation $\sum_{i=0}^{n} c_i x_i = c$ with c_i even and c odd, it is clear that the least significant bit of the sum is always zero. Thus, the equation is unsatisfiable which can be detected after reading the right-most input vector. Let us first consider the translation of equations. Given a term $T = \left(\sum_{i=0}^{n} c_i x_i\right) - c$, the translation is based on the following recursion ($T \cong_k 0$ holds iff T is divisible by k):

$$T = 0 \Leftrightarrow T \cong_2 0 \wedge \lfloor T/2 \rfloor = 0 \tag{1}$$

Unwinding this equation is essentially equivalent to checking whether the bits of the sum (first conjunct) and the carry (second conjunct) are zero. Hence, by reading the input from right to left, we do not use the states of an AFA to store the current value of the sum, but to store the result of the division, i.e., the carry.

Before we can construct an AFA for an equation or inequality, we must determine the number of required state variables (since in [5] the states are represented explicitly instead of symbolically, they can be constructed on-the-fly). For that purpose, we have to compute the maximal (minimal) carry that can occur while reading a word.

Theorem 1. *Given a relation $\sum_{i=0}^{n} c_i x_i \bowtie c$ with $\bowtie \in \{=, <\}$, let $c_{\max}$ and $c_{\min}$ denote the sum of the positive and negative coefficients, respectively. Then, the following holds for the maximal carry $k_{\max}$ and the minimal carry $k_{\min}$:*

$$k_{\max} = \begin{cases} c_{\max} - 1 & \textit{if } c_{\max} + c > 1 \\ -c & \textit{otherwise} \end{cases} \qquad k_{\min} = \begin{cases} c_{\min} & \textit{if } c_{\min} + c < 0 \\ -c & \textit{otherwise} \end{cases}$$

Proof. In the worst case, either only variables with positive or with negative coefficients contribute to the carry. Hence, the sequence of maximal (minimal) carries is $x_{i+1} := f(x_i)$ starting with $x_0 := -c$, where $f(x) = \lfloor (x + A)/2 \rfloor$ for $A = c_{\max}$ ($A = c_{\min}$). Note that f is monotonic, since it is composed of the monotonic functions $\lambda x.x + A$, $\lambda x.x/2$, and $\lambda x.\lfloor x \rfloor$. Thus, the sequence is monotonically increasing if $x_0 \leq f(x_0)$ and monotonically decreasing if $x_0 \geq f(x_0)$. Moreover, f has two fixpoints, namely a greatest fixpoint $\nu x.f = A$ and a least fixpoint $\mu x.f = A - 1$. In order to determine

$k_{\max} := \max\{x_i \mid i \geq 0\}$, we have to distinguish between two cases: If $x_0 < f(x_0)$, i.e., $-c < \lfloor(-c + c_{\max})/2\rfloor \Leftrightarrow c_{\max} + c > 1$, then $\lim_{i\to\infty} x_i = \mu x.f = A - 1$, and hence, $k_{\max} = c_{\max} - 1$. Otherwise, the sequence is monotonically decreasing and converges to A so that $k_{\max} = x_0 = -c$. The proof for $k_{\min}$ is analog. □

Thus, the number of bits required to store the carries is $m := \max(||k_{\max}||, ||k_{\min}||)$, where $||.|| : \mathbb{Z} \to \mathbb{N}$ yields the number of bits required to represent an integer in two's complement encoding.

Definition 4 (Translating Equations to AFAs). *Given a set of integer variables $\mathcal{V}_\mathbb{Z} = \{x_0, \ldots, x_n\}$, let $V := \{v_0, \ldots, v_n\}$ be the set of input variables. Then, an equation $\sum_{i=0}^{n} c_i x_i = c$ is translated to an AFA (Q, V, δ, I, F) with $m + 1$ state variables $Q = \{q_0, \ldots, q_m\}$ such that the following holds, where $d := \sum_{i=0}^{n} c_i v_i$:*

- $\langle\delta(q_m), \ldots, \delta(q_1)\rangle_\mathbb{Z} = \langle q_m, \ldots, q_1\rangle_\mathbb{Z} + \lfloor d/2\rfloor$
- $\delta(q_0) \equiv (q_0 \wedge d \cong_2 0)$
- $I \equiv (\langle q_m, \ldots, q_1\rangle_\mathbb{Z} - \lfloor d/2\rfloor = 0 \wedge q_0)$
- $\langle F(q_m), \ldots, F(q_1)\rangle_\mathbb{Z} = -c$
- $F(q_0) = 1$

An AFA constructed according to the above definition essentially implements the successive application of Equation (1). Given a word with k letters, we obtain the following formula by unwinding the equation, where q_0 represents the conjunction of the sum bits at positions $0 \leq i < k$ and $\langle q_m, \ldots, q_1\rangle_\mathbb{Z}$ the carry at step k:

$$\underbrace{T \cong_2 0 \wedge (T/2) \cong_2 0 \wedge \cdots \wedge (T/2^{k-1}) \cong_2 0}_{q_0} \wedge \underbrace{\lfloor T/2^k \rfloor}_{\langle q_m,\ldots,q_1\rangle_\mathbb{Z}} = 0$$

As an example, consider the equation $2x - y = 1$. With $c_{\max} = 2$ and $c_{\min} = -1$ we obtain $k_{\max} = 1$ and $k_{\min} = -1$. Since $m = \max(||1||, ||-1||) = \max(2, 1) = 2$, a total number of three state bits are required. The reachable part of the corresponding DFA is shown in Figure 1, where dotted transitions indicate the application of the initial formula. Note that this leads to nondeterministic behavior, since one might apply the initial formula, but one might also unwind the AFA once more. However, the remaining (large) part is always deterministic.

The translation of inequalities to AFAs is very similar to the translation of equations except that we do not have to check whether all bits of the sum are zero. It suffices to check whether the carry will eventually be negative. This can be easily done by examining the most significant bit which determines the sign in two's complement encoding.

Definition 5 (Translating Inequalities to AFAs). *Given a set of integer variables $\mathcal{V}_\mathbb{Z} = \{x_0, \ldots, x_n\}$, let $V := \{v_0, \ldots, v_n\}$ be the set of input variables. Then, an inequality $\sum_{i=0}^{n} c_i x_i < c$ is translated to an AFA (Q, V, δ, I, F) with m state variables $Q = \{q_1, \ldots, q_m\}$ such that the following holds, where $d := \sum_{i=0}^{n} c_i v_i$:*

- $\langle\delta(q_m), \ldots, \delta(q_1)\rangle_\mathbb{Z} = \langle q_m, \ldots, q_1\rangle_\mathbb{Z} + \lfloor d/2\rfloor$
- $I \equiv (\langle q_m, \ldots, q_1\rangle_\mathbb{Z} - \lfloor d/2\rfloor < 0)$
- $\langle F(q_m), \ldots, F(q_1)\rangle_\mathbb{Z} = -c$

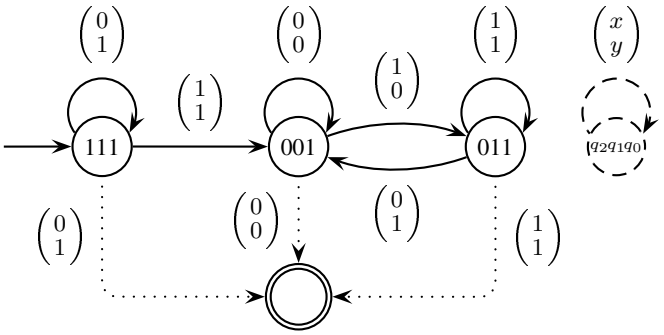

Fig. 1. Automaton for the equation $2x - y = 1$

4 Checking Emptiness of AFAs

Once we have constructed the AFA $\mathcal{A}_\varphi$ for a QFPAbit formula φ, checking satisfiability of φ amounts to checking whether the language of $\mathcal{A}_\varphi$ is empty. If so, there does not exist a satisfying assignment for φ. A straightforward way to check emptiness of an AFA is to unwind it according to Definition 2 without assigning values to the input variables. Hence, a formula is satisfiable iff there exists a $k \geq 0$ such that the formula obtained by unwinding the AFA is satisfiable. However, this only yields a semi-decision procedure that cannot be used directly to prove that a formula is unsatisfiable, since this would require infinitely many unwinding steps.

To solve this problem, we could make use of the fact that for finite state systems there always exists an upper bound on the required number of unwinding steps which is referred to as the completeness threshold in bounded model checking. In general, however, computing the completeness threshold is a nontrivial task and does not always yield a tight bound. On the other hand, regarding the special case of quantifier-free Presburger formulas, it is well-known that if a formula has a satisfying solution, there is one whose size, measured in the number of bits, is polynomially bounded in the size of the formula. This allows one to translate every quantifier-free Presburger formula to an equisatisfiable propositional formula of polynomial size [8] (in theory, this follows from the fact that deciding formulas of linear integer arithmetic is NP-complete [28]).

Unfortunately, computing an upper bound on the number of unwinding steps for checking emptiness of an AFA is hardly feasible in practice. This is due to the fact that QFPAbit is more expressive than quantifier-free Presburger arithmetic (cf. Section 2.2). In particular, if a formula contains bitvector operations, there may not even exist a polynomially sized model w.r.t. to the length of the formula. To prove this, let R_k be a family of formulas defined as follows with $k \geq 2$:

$$R_k := x > 0 \wedge \mathsf{pow2}\left(1 + \overrightarrow{\bigvee_{i=0}^{k-1}} 2^i x\right) \wedge \bigwedge_{i=0}^{k-2} \bigwedge_{j=i+1}^{k-1} 2^i x \overrightarrow{\wedge} 2^j x = 0$$

Note that a formula $\mathsf{pow2}(f(x))$ can be easily translated to the equisatisfiable formula $\mathsf{pow2}(y) \wedge y = f(x)$. The models of R_k are characterized by the following lemma:

Lemma 1. *A natural number x is a model of R_k iff every k-th bit of x is one and all other bits are zero.*

Proof. The operands of the bitvector operations are multiples of x, each shifted by i bits to the left for $0 \leq i < k$, as shown in the following diagram:

$$\begin{array}{rcccccccccccc}
x & & & & & 0 & x_n & \dots & x_k & x_{k-1} & \dots & \dots & x_1 & x_0 \\
2x & & & & 0 & x_n & \dots & x_k & x_{k-1} & \dots & \dots & x_1 & x_0 & 0 \\
 & & & & & & & & \vdots & & & & & \\
2^{k-1}x & 0 & x_n & \dots & x_k & x_{k-1} & \dots & \dots & x_1 & x_0 & \dots & 0 & 0 & 0
\end{array}$$

First, we prove that a number $x = \langle 0, x_n, \dots, x_0 \rangle_{\mathbb{Z}}$ is a model of R_k if $x_i = 1$ and $x_j = 0$ for all $0 \leq i, j \leq n$ such that $i \cong_k 0$ and $j \neq i$ holds. The first part of the conjunction is obviously satisfied, since the most significant bit of x is zero. The second part is also satisfied, since each column in the above diagram contains an x_i with $i \cong_k 0$. Thus, the result of the bitvector disjunction is a string of 1's and the increment of the corresponding number is a power of two. The third part of the conjunction performs a pairwise comparison of the rows to ensure that at most one bit is set in each column. Since the columns consist of exactly k elements, they cannot contain an x_i and an x_{i+k} with $i \cong_k 0$. Hence, this part is also satisfied. The reverse direction of the proof follows from the fact that this is the only assignment of the x_i's that satisfy the formula. Given an arbitrary x, we must have $x_0 = 1$ by the second part of the conjunction, $x_1 = \dots = x_{k-1} = 0$ by the third part, and so on. □

Theorem 2. *Given a satisfiable* QFPAbit *formula φ, the size of its smallest model, measured in the number of bits, is not polynomially bounded in the size of φ.*

Proof. Let $|\varphi|$ denote the size of a formula φ, where the size of a constant is measured in its number of bits. Then, we have $|R_k| \in O(k^3)$. Moreover, given that $p : \mathbb{N} \to \mathbb{N}$ is the sequence of prime numbers, it holds that $p(i) \sim i \cdot \ln i$ and $\prod_{i=1}^{n} p(i) \sim e^n$ [29]. We define $S_n := -2 \overrightarrow{\wedge} \bigwedge_{i=1}^{n} R_{p(i)}$ (the bitvector conjunction with -2 masks out the least significant bits) and obtain $|S_n| \in O(\sum_{i=1}^{n} (i \cdot \ln i)^3)$. Hence, the size of S_n is polynomially bounded by $O(n^4 (\ln n)^3)$. Note that a number x satisfies S_n iff every l-th bit of x is true and all other bits are zero, where $l = \prod_{i=1}^{n} p(i)$. Thus, the size of the smallest solution is $\Theta(e^n)$ which is not polynomially bounded by n. □

For this reason, we use another approach which has been originally proposed for checking safety properties in bounded model checking [30]. As a major advantage, this approach uses an inductive style of reasoning and terminates as soon as possible, i.e., it does not suffer from non-tight bounds. Given a finite set of states $\mathcal{S}$, a complete transition relation $\mathcal{R} \subseteq S \times S$, and a set of initial states $\mathcal{I} \subseteq S$, let $\mathsf{path}(s_0 \dots s_k)$ hold iff $(s_i, s_{i+1}) \in \mathcal{R}$ for all $0 \leq i < k$. Moreover, let $\mathsf{loopFree}(s_0 \dots s_k)$ hold iff $\mathsf{path}(s_0 \dots s_k)$ holds and $s_i \neq s_j$ for all $0 \leq i < j \leq k$. Then, the method presented in [30] aims at checking whether a property $\mathcal{P} \subseteq \mathcal{S}$ invariantly holds on all paths originating in $\mathcal{I}$ using the following induction scheme, where k denotes the induction depth:

$$\begin{array}{ll}
\textit{base case:} & s_0 \in \mathcal{I} \wedge \mathsf{path}(s_0 \dots s_k) \rightarrow (\forall i. 0 \leq i \leq k \rightarrow s_i \in \mathcal{P}) \\
\textit{induction step:} & \mathsf{loopFree}(s_0 \dots s_{k+1}) \wedge (\forall i. 0 \leq i \leq k \rightarrow s_i \in \mathcal{P}) \rightarrow s_{k+1} \in \mathcal{P}
\end{array}$$

The base case checks whether $\mathcal{P}$ holds on a path $s_0 \dots s_k$, and the induction step checks whether this path can be extended without violating $\mathcal{P}$. Thus, $\mathcal{P}$ invariantly holds on all paths if both conditions are valid[3]. Note that the method is sound and complete for sufficiently large values of k (completeness follows simply speaking from the fact that every infinite path in a finite state system eventually contains a loop).

In order to prove that an AFA (Q, V, δ, I, F) is empty using induction, we show that the initial formula I never evaluates to true for the given assignment of the final states F, i.e., we set $\mathcal{P} := \neg I$. For that purpose, the AFA is iteratively unwound until the base case fails or the induction step holds. As mentioned previously, this does not require the introduction of new state variables due to the equational structure of AFAs. Instead, it suffices to replace the input variables with new variables at each step, since a path is uniquely determined by the read word. As a consequence, the size of the resulting formulas is reduced significantly which simplifies the check for satisfiability.

The algorithm for checking emptiness of an AFA is shown in Figure 2, where a set of paths is viewed as a tuple of propositional formulas. The current set of paths and their prefixes are stored in the list unwind. Remembering the suffixes is necessary to update the variable loopFree when a path is extended. The variable reject corresponds to the formula $\forall i.0 \leq i \leq k \rightarrow s_i \in \mathcal{P}$ of the induction scheme, where $\mathcal{P} := \neg I$. That is, reject holds iff a path and its suffixes are nonaccepting. As the first step, the algorithm checks whether I is unsatisfiable. If so, the AFA is clearly empty. Otherwise, induction is applied for increasing depths, starting with paths of length one. If the base case fails, a counterexample has been found and the algorithm returns false. Otherwise, the current set of paths is extended and the induction step is checked. If the induction step also holds, it follows that the AFA is empty. If it does not hold, the procedure is repeated for an increased induction depth.

5 Experimental Results

The approach presented in this paper has been implemented in our symbolic model checker Beryl which is part of the Averest framework[4]. We performed two sets of experiments, one with a number of benchmarks contained in Averest and one with the quantifier-free linear integer arithmetic (QF_LIA) benchmarks of the satisfiability modulo theories library (SMT-LIB) [31]. The former are given in our synchronous language Quartz and were compiled to symbolically encoded transition systems. As the base logics, our compiler supports propositional logic as well as Presburger arithmetic with bitvector operations. The generated transition systems are essentially the same for both logics, except that arithmetic operations on integers are translated to the corresponding operations on fixed sized bitvectors when using propositional logic as the base logic.

The results for the Averest benchmarks are shown in Table 1. For each benchmark we proved a liveness property (first row) and disproved a safety property (second row). All runtimes are given in seconds and were measured on a Xeon processor with 3GHz.

[3] As described in [30], there exists a dual type of induction scheme that can be thought of as working in the backward direction. For the sake of simplicity, however, we restrict ourselves to the forward case. Nevertheless, our implementation supports both approaches.

[4] `http://www.averest.org`

```
function empty(Q, V, δ, I, F)
  if ¬sat(I) then return true;
  reject := ¬I;
  unwind := cons((q_0, ..., q_m), []);
  loopFree := true;
  loop
    // base case
    if ¬valid(reject[q_i/F(q_i)]^{0≤i≤m}) then return false;
    // unwind
    (w_0, ..., w_n) := createFreshVariables(n + 1);
    (r_0, ..., r_m) := head(unwind);
    (s_0, ..., s_m) := (δ(q_0)[q_i/r_i, v_j/w_j]_{0≤j≤n}^{0≤i≤m}, ..., δ(q_m)[q_i/r_i, v_j/w_j]_{0≤j≤n}^{0≤i≤m});
    u := unwind;
    repeat
      (t_0, ..., t_m) := head(u);
      loopFree := loopFree ∧ ⋁_{i=0}^{m} s_i ⊕ t_i;
      u := tail(u);
    until u = [];
    unwind := cons((s_0, ..., s_m), unwind);
    rejectNext := reject[q_i/s_i]^{0≤i≤m};
    // induction step
    if valid(loopFree ∧ reject → rejectNext) then return true;
    reject := reject ∧ rejectNext;
  end;
end;
```

Fig. 2. Algorithm for checking emptiness of an AFA using induction

A dash indicates that a benchmark could not be solved in less than ten minutes. The first two columns show the runtimes for bounded model checking (BMC) using AFAs and DFAs. The latter were constructed by the method presented in [5] and use a semi-symbolic encoding, i.e., the states are represented explicitly and the transitions symbolically by means of BDDs. Moreover, we measured the runtimes for global model checking (GMC) using DFAs (GMC is not possible for AFAs, since they do not support image computation). As can be seen, AFAs are much more efficient than DFAs for BMC, and in many cases also more efficient than GMC/DFA. For BubbleSort, however, BMC/AFA is significantly slower which is due to the fact that this benchmark requires a high bound in BMC. Finally, we verified the benchmarks using Cadence SMV and NuSMV for integers with fixed bitwidths (dynamic variable reordering was enabled). One might argue that such a comparison is like comparing apples and oranges, particularly since the results for AFAs are based on bounded model checking, while the results for SMV/NuSMV were obtained using global model checking[5]. Nevertheless,

[5] Unfortunately, neither SMV nor NuSMV was able to check the specifications generated by our compiler using bounded model checking, even though the specifications are simple safety and liveness properties.

Table 1. Runtimes for Averest benchmarks

Benchmark	Beryl			NuSMV				SMV			
	BMC AFA	BMC DFA	GMC DFA	8 bit	16 bit	24 bit	32 bit	8 bit	16 bit	24 bit	32 bit
BinarySearch	0,3	-	0,4	16,7	24,9	132,8	182,9	2,3	12,1	52,3	99,8
	0,2	22,4	1,0	18,7	148,1	-	-	12,6	-	-	-
BubbleSort	117,3	-	1,3	3,1	27,7	104,7	347,0	0,1	0,1	0,1	0,1
	102,6	-	19,5	9,6	-	-	-	2,4	77,6	45,3	93,3
FastMax	0,1	0,8	0,2	1,3	26,9	47,2	398,9	0,0	0,0	0,1	0,1
	0,3	1,8	1,8	3,6	-	-	-	-	-	-	-
LinearSearch	0,4	329,7	0,2	1,4	6,9	20,3	34,0	0,9	7,7	15,6	38,9
	0,1	0,1	0,3	2,2	11,8	26,1	58,1	2,2	8,4	19,4	40,8
MinMax	0,6	-	1,5	5,6	254,1	-	-	0,0	0,0	0,1	0,1
	0,3	148,9	71,4	92,4	-	-	-	104,3	-	-	-
ParallelPrefixSum	1,3	-	39,6	4,1	37,5	235,9	-	0,1	0,1	0,1	0,1
	7,3	-	-	-	-	-	-	453,7	266,5	-	-
ParallelSearch	0,2	-	8,9	1,4	8,7	11,0	40,6	1,1	1,8	5,7	29,7
	60,6	541,7	8,6	3,7	22,4	17,6	37,1	1,9	3,7	11,1	19,5
Partition	1,1	-	14,0	147,0	-	-	-	39,7	208,2	-	-
	0,4	-	116,6	256,1	-	-	-	112,1	-	-	-
SortingNetwork4	0,2	149,7	0,3	1,0	35,5	143,1	344,8	0,0	0,0	0,0	0,1
	0,1	174,7	0,9	74,5	-	-	-	-	243,5	-	-
SortingNetwork8	3,0	-	-	534,2	-	-	-	0,0	0,1	0,1	0,1
	2,1	-	-	-	-	-	-	-	-	-	-

we list the results to compare our approach with sophisticated model checkers that are frequently used in hardware verification.

The results for the SMT-LIB benchmarks are shown in Table 2, where we compared Beryl with CVC Lite 2.5[6], MathSat 3.3.1[7], and Yices 0.2[8]. We list only those benchmarks that could be solved by at least one of the tools within five minutes. As can be seen, the runtimes largely differ depending on the benchmark. For example, our approach clearly outperforms all other tools for the SIMPLEBIT_ADDER benchmarks that could be solved up to size 10 using Beryl, whereas the other tools could only solve them for size 5 (CVC Lite), 7 (MathSat), and 8 (Yices). For the FISCHER benchmarks, however, the situation is converse. These benchmarks are most efficiently solved using MathSat and Yices. For the remaining benchmarks, Beryl can in many cases compete with the best tool and is usually much faster than the slowest tool.

To check satisfiability of a formula, our implementation currently constructs an AFA $\mathcal{A}$ for the whole formula and then checks whether $\mathcal{A}$ is empty. Clearly, this is more than necessary if the result only depends on some subformulas. A better approach is to check subformulas lazily, i.e., by need. For that purpose, MathSat and Yices use an extension of the DPLL procedure for propositional logic that often allows one to restrict the search for a satisfying assignment to a small number of subformulas. As an example, given the formula $x \geq 0 \wedge (p \vee x + y = z)$, it suffices to prove that $x \geq 0$ is satisfiable, provided

[6] `http://www.cs.nyu.edu/acsys/cvcl/`
[7] `http://mathsat.itc.it/`
[8] `http://yices.csl.sri.com/`

Table 2. Runtimes for SMT-LIB benchmarks

Benchmark	Status	Beryl	CVC Lite	MathSat	Yices
ckt_PROP0_tf_15	sat	< 1	24.9	42.2	< 1
ckt_PROP0_tf_20	sat	< 1	-	66.0	< 1
FISCHER5-2-fair	unsat	20.1	1.3	< 1	< 1
FISCHER5-3-fair	unsat	2.3	2.9	< 1	< 1
FISCHER5-4-fair	unsat	4.2	5.4	< 1	< 1
FISCHER5-5-fair	unsat	96.9	9.6	< 1	< 1
FISCHER5-6-fair	unsat	-	15.1	< 1	< 1
FISCHER5-7-fair	unsat	-	29.5	< 1	< 1
FISCHER5-8-fair	unsat	-	67.6	1.9	< 1
FISCHER5-9-fair	unsat	-	116.4	4.1	< 1
FISCHER5-10-fair	sat	38.0	-	1.8	< 1
MULTIPLIER_2	unsat	< 1	2.3	< 1	< 1
MULTIPLIER_3	unsat	< 1	32.1	< 1	< 1
MULTIPLIER_4	unsat	< 1	172.9	1.2	< 1
MULTIPLIER_5	unsat	1.3	-	7.1	< 1
MULTIPLIER_6	unsat	5.6	-	42.1	1.0
MULTIPLIER_7	unsat	33.7	-	255.4	10.3
MULTIPLIER_8	unsat	-	-	-	54.3
MULTIPLIER_64	sat	4.3	-	< 1	< 1
MULTIPLIER_PRIME_2	sat	< 1	abort	< 1	< 1
MULTIPLIER_PRIME_3	sat	< 1	segfault	< 1	< 1
MULTIPLIER_PRIME_4	sat	< 1	wrong	< 1	< 1
MULTIPLIER_PRIME_5	sat	< 1	abort	< 1	< 1
MULTIPLIER_PRIME_6	sat	< 1	-	< 1	< 1
MULTIPLIER_PRIME_7	sat	< 1	segfault	< 1	< 1
MULTIPLIER_PRIME_8	sat	< 1	-	< 1	< 1
MULTIPLIER_PRIME_9	sat	2.2	-	< 1	< 1
MULTIPLIER_PRIME_10	sat	2.6	-	< 1	< 1
MULTIPLIER_PRIME_11	sat	3.6	-	< 1	< 1
MULTIPLIER_PRIME_12	sat	4.8	-	1.0	< 1
MULTIPLIER_PRIME_13	sat	9.9	segfault	3.2	< 1
MULTIPLIER_PRIME_14	sat	25.6	-	6.2	< 1
MULTIPLIER_PRIME_15	sat	30.6	-	21.8	< 1
MULTIPLIER_PRIME_16	sat	16.1	-	67.3	< 1
MULTIPLIER_PRIME_32	sat	2.5	segfault	< 1	< 1
MULTIPLIER_PRIME_64	sat	4.3	-	< 1	< 1
SIMPLEBITADDER_COMPOSE_2	unsat	< 1	< 1	< 1	< 1
SIMPLEBITADDER_COMPOSE_3	unsat	< 1	3.8	< 1	< 1
SIMPLEBITADDER_COMPOSE_4	unsat	< 1	51.6	1.1	< 1
SIMPLEBITADDER_COMPOSE_5	unsat	< 1	70.0	5.3	< 1
SIMPLEBITADDER_COMPOSE_6	unsat	1.2	-	26.9	< 1
SIMPLEBITADDER_COMPOSE_7	unsat	7.2	-	231.3	5.2
SIMPLEBITADDER_COMPOSE_8	unsat	23.7	-	-	94.3
SIMPLEBITADDER_COMPOSE_9	unsat	56.2	-	-	-
SIMPLEBITADDER_COMPOSE_10	unsat	115.0	-	-	-
wisa1	sat	3.4	8.8	223.5	< 1
wisa2	unsat	-	113.8	-	< 1
wisa3	sat	6.4	226.0	-	< 1
wisa4	sat	4.1	28.0	-	1.1
wisa5	unsat	-	-	-	6.1

that p has already been set to true. Of course, such a lazy decision procedure can also be used with the approach presented in this paper to check arithmetic and bitvector formulas on demand.

6 Summary and Conclusion

We proposed a decision procedure for quantifier-free Presburger arithmetic with bitvector operations. The translation to alternating automata allows us to benefit from efficient SAT solvers for checking emptiness of the automata using induction. This is necessary, since formulas of the considered logic cannot always be reduced to a propositional formula of polynomial size. The experimental results show that our approach can compete with state-of-the-art decision procedures and is sometimes even more efficient. We plan to combine the use of AFAs with quantifier elimination in order to support quantified Presburger arithmetic. For that purpose, the translation has to be extended in order to deal with congruences that occur during quantifier elimination.

References

1. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In Leja, F., ed.: Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich, Warszawa 1929 (Comptes–rendus du I Congrès des Mathématiciens des Pays Slaves, Varsovie 1929), Warszawa (1929) 92–101 (supplement on p. 395)
2. Lahiri, S., Seshia, S., Bryant, R.: Modeling and verification of out-of-order microprocessors in UCLID. In Aagaard, M., O'Leary, J., eds.: Conference on Formal Methods in Computer Aided Design (FMCAD). Volume 2517 of LNCS., Portland, USA, Springer (2002) 142–159
3. Büchi, J.: Weak second order arithmetic and finite automata. Z. Math. Logik Grundlagen Math. **6** (1960) 66–92
4. Boudet, A., Comon, H.: Diophantine equations, Presburger arithmetic and finite automata. In Kirchner, H., ed.: Colloquium on Trees in Algebra and Programming (CAAP). Volume 1059 of LNCS., Linköping, Sweden, Springer (1996) 30–43
5. Wolper, P., Boigelot, B.: On the construction of automata from linear arithmetic constraints. In Graf, S., Schwartzbach, M., eds.: Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 1785 of LNCS., Berlin, Germany, Springer (2000) 1–19
6. Klaedtke, F.: On the automata size for Presburger arithmetic. Technical Report 186, Institute of Computer Science at Freiburg University (2003)
7. Ganesh, V., Berezin, S., Dill, D.: Deciding Presburger arithmetic by model checking and comparisons with other methods. In Aagaard, M., O'Leary, J., eds.: Conference on Formal Methods in Computer Aided Design (FMCAD). Volume 2517 of LNCS., Portland, USA, Springer (2002) 171–186
8. Seshia, S., Bryant, R.: Deciding quantifier-free Presburger formulas using parameterized solution bounds. Logical Methods in Computer Science **1**(2:6) (2005) 1–26
9. Boigelot, B.: The Liège automata-based symbolic handler (LASH) (2006) `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
10. Janicic, P., Green, I., Bundy, A.: A comparison of decision procedures in Presburger arithmetic. Research Paper 872, University of Edinburgh (1997)

11. Shiple, T., Kukula, J., Ranjan, R.: A comparison of Presburger engines for EFSM reachability. In Hu, A., Vardi, M., eds.: Conference on Computer Aided Verification (CAV). Volume 1427 of LNCS., Vancouver, BC, Canada, Springer (1998) 280–292
12. Schuele, T., Schneider, K.: Symbolic model checking by automata based set representation. In Ruf, J., ed.: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Tübingen, Germany, GI/ITG/GMM, Shaker (2002) 229–238
13. Strichman, O.: On solving Presburger and linear arithmetic with SAT. In Aagaard, M., O'Leary, J., eds.: Conference on Formal Methods in Computer Aided Design (FMCAD). Volume 2517 of LNCS., Portland, USA, Springer (2002) 160–170
14. Kroening, D., Ouaknine, J., Seshia, S., Strichman, O.: Abstraction-based satisfiability solving of Presburger arithmetic. In Alur, R., Peled, D., eds.: Conference on Computer Aided Verification (CAV). Volume 3114 of LNCS., Boston, MA, USA, Springer (2004) 308–320
15. Yu, S.: Regular languages. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Volume 1. Springer (1997) 41–110
16. Fellah, A., Jürgensen, H., Yu, S.: Constructions for alternating finite automata. International Journal of Computer Mathematics **35** (1990) 117–132
17. Fellah, A.: Equations and regular-like expressions for AFA. International Journal of Computer Mathematics **51** (1994) 157–172
18. Chandra, A., Kozen, D., Stockmeyer, L.: Alternation. Journal of the ACM **28**(1) (1981) 114–133
19. Brzozowski, J., Leiss, E.: On equations for regular languages, finite automata, and sequential networks. Theoretical Computer Science **10** (1980) 19–35
20. Leiss, E.: Succinct representation of regular languages by Boolean automata. Theoretical Computer Science **13** (1981) 323–330
21. Leiss, E.: Succinct representation of regular languages by Boolean automata II. Theoretical Computer Science **38** (1985) 133–136
22. Huerter, S., Salomaa, K., Wu, X., Yu, S.: Implementing reversed alternating finite automaton (r-AFA) operations. In Champarnaud, J.M., Maurel, D., Ziadi, D., eds.: International Workshop on Implementating Automata (WIA). Volume 1660 of LNCS., Rouen, France, Springer (1999) 69–81
23. Salomaa, K., Wu, X., Yu, S.: Efficient implementation of regular languages using r-AFA. In Wood, D., Yu, S., eds.: International Workshop on Implementing Automata (WIA). Volume 1436 of LNCS., London, Ontario, Canada, Springer (1998) 176–184
24. Salomaa, K., Wu, X., Yu, S.: Efficient implementation of regular languages using reversed alternating finite automata. Theoretical Computer Science **231**(1) (2000) 103–111
25. Tuerk, T., Schneider, K.: Relationship between alternating omega-automata and symbolically represented nondeterministic omega-automata. Internal Report 340, Department of Computer Science, University of Kaiserslautern, `http://kluedo.ub.uni-kl.de` (2005)
26. Bès, A.: A survey of arithmetical definability. Bulletin of the Société Mathématique Belgique (2002) 1–54
27. Bruyere, V., Hansel, G., Michaux, C., Villemaire, R.: Logic and p-recognizable sets of integers. Bulletin of the Société Mathématique Belgique **1** (1994) 191–238
28. von zur Gathen, J., Sieveking, M.: A bound on solutions of linear integer equalities and inequalities. Proceedings of the American Mathematical Society **72**(1) (1978) 155–158
29. Hardy, G., Wright, E.: An introduction to the theory of numbers. Oxford University Press (1979)
30. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In Hunt, W., Johnson, S., eds.: Conference on Formal Methods in Computer Aided Design (FMCAD). Volume 1954 of LNCS., Austin, Texas, USA, Springer (2000) 108–125
31. Ranise, S., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2006) `http://goedel.cs.uiowa.edu/smtlib`.