Deciding Presburger Arithmetic by Model Checking and Comparisons with Other Methods *

Vijay Ganesh, Sergey Berezin, and David L. Dill

Stanford University
{vganesh,berezin, dill}@stanford.edu

Abstract. We present a new way of using Binary Decision Diagrams in automata based algorithms for solving the satisfiability problem of quantifier-free Presburger arithmetic. Unlike in previous approaches [5, 2, 19], we translate the satisfiability problem into a model checking problem and use the existing BDD-based model checker SMV [13] as our primary engine.
We also compare the performance of various Presburger tools, based on both automata and ILP approaches, on a large suite of parameterized randomly generated test cases. The strengths and weaknesses of each approach as a function of these parameters are reported, and the reasons for the same are discussed. The results show that no single tool performs better than the others for all the parameters.
On the theoretical side, we provide tighter bounds on the number of states of the automata.

1 Introduction

Efficient decision procedures for logical theories can greatly help in the verification of programs or hardware designs. For instance, quantifier-free Presburger arithmetic [15] has been used in RTL-datapath verification [3], and symbolic timing verification [1].[1] However, the satisfiability problem for the quantifier free fragment is known to be NP-complete [14]. Consequently, the search for practically efficient algorithms becomes very important.

Presburger arithmetic is defined to be the first-order theory of the structure

$$\langle \mathbb{Z}, 0, \leq, + \rangle,$$

where $\mathbb{Z}$ is the set of integers. The satisfiability of Presburger arithmetic was shown to be decidable by Presburger in 1927 [15, 12]. This theory is usually defined over the natural numbers $\mathbb{N}$, but can easily be extended to the integers (which is important for practical applications) by representing any integer variable x by two natural variables: $x = x_+ - x_-$. This reduction obviously has no effect on known decidability or complexity results.

* This research was supported by GSRC contract SA2206-23106PG-2 and in part by National Science Foundation CCR-9806889-002. The content of this paper does not necessarily reflect the position or the policy of GSRC, NSF, or the Government, and no official endorsement should be inferred.

[1] In [1] Presburger formulas have quantifiers, but without alternation, and therefore, are easy to convert into quantifier-free formulas.

The remainder of the paper focuses on quantifier-free Presburger arithmetic because many verification problems do not require quantification, and because the performance of decision procedures on quantifier-free formulas may be qualitatively different from the quantified case. This paper has two primary goals: presentation of a new decision procedure based on model checking and comparison of the various approaches to deciding quantifier-free Presburger arithmetic and their implementations.

There are three distinct ways of solving the satisfiability problem of quantifier-free Presburger, namely the Cooper's method [8], the integer linear programming (ILP) based approaches, and the automata-based methods. Cooper's method is based on Presburger's original method for solving quantified formulas, only more efficient. Using Cooper's method on a quantifier-free formula still requires introducing existential quantifiers and then eliminating them. This process results in an explosion of new atomic formulas, so the method is probably too inefficient to be competitive with other approaches.

Since atomic formulas are linear integer equalities and inequalities, it is natural to think of the *integer linear programming* (ILP) algorithms as a means to determine the satisfiability of quantifier-free formulas in Presburger arithmetic. ILP algorithms maximize an objective function, subject to constraints in the form of a conjunction of linear equalities and inequalities. Along the way, the system is checked for satisfiability (usually called *feasibility*), which is the problem of interest in this paper.

There are many efficient implementations of ILP solvers available. We have experimented with the commercial tool CPLEX and open source implementations LP_SOLVE and OMEGA [16]. The OMEGA tool is specifically tuned to solve integer problems, and is an extension of the Fourier-Motzkin linear programming algorithm [9] to integers [18]. In order to solve an arbitrary quantifier-free formula, it must first be converted to *disjunctive normal form* (DNF), then ILP must be applied to each disjunct until a satisfiable one is found. If any of the disjuncts is satisfiable, then the entire formula is satisfiable. This conversion to DNF may lead to an exponential explosion of the formula size.

In addition, unlike automata methods, the existing implementations lack the support for arbitrarily large integers and use native machine arithmetic. This has two consequences. Firstly, it obstructs making a fair comparison of the ILP tools with automata methods, since the two are not feature equivalent. The use of native machine arithmetic by ILP tools gives them an unfair performance advantage. Secondly, the support for large integers may be crucial in certain hardware verification problems, where the solution set may have integers larger than the *int* types supported natively by the hardware. For instance, many current RTL-datapath verification approaches use ILP [11, 3], but these approaches cannot be scaled with the bit-vector size in the designs.

A third approach uses finite automata theory. The idea that an atomic Presburger formula can be represented by a finite-state automaton goes back at least to Büchi [5]. Boudet and Comon [2] proposed a more efficient encoding than Büchi's. Later, Wolper and Boigelot [19] further improved the method of Boudet and Comon and implemented the technique in the system called LASH. Another automata-based approach is to translate the atomic formulas into WS1S (weak monadic second order logic with one suc-

cessor) and then use the MONA tool [10]. MONA is a decision procedure for WS1S and uses *Binary Decision Diagrams* (BDDs, [4]) internally to represent automata.

In this paper, a new automata-based approach using symbolic model checking [7] is proposed and evaluated. The key idea is to convert the quantifier-free Presburger formula into a sequential circuit which is then model checked using SMV [13]. Experiments indicate that the SMV approach is quite efficient and more scalable on formulas with large coefficients than all the other automata-based techniques. The reason for this is the use of BDDs to represent *both the states and the transitions* of the resulting automaton. Another factor which contributes to the efficiency is that SMV uses a highly optimized BDD package. In addition, the use of an existing tool saves a lot of implementation effort. The experiments required only a relatively small Perl script to convert Presburger formulas into the SMV language.

The other tools do not use BDDs for the states because they perform quantifier elimination by manipulating the automata directly. Namely, each quantifier alternation requires projection and determinization of the automaton. The use of BDDs for the states can make the implementation of the determinization step particularly hard.

We also compare various automata and ILP-based approaches on a suite of 400 randomly generated Presburger formulas. The random generation was controlled by several parameters, such as the number of atomic formulas, the number of variables, and maximum coefficient size. For every approach we identify classes of Presburger formulas for which it either performs very poorly or very efficiently. Only one similar comparison has been done previously in [17]. However, their examples consist of a rather small set of quantified Presburger formulas obtained from real hardware verification problems. The goal of our comparison is to study the performance trends of various approaches and tools depending on different parameters of quantifier-free Presburger formulas.

The paper is organized as follows. Section 2 explains the automata construction algorithms which are the same as in [19, 2], except for the tighter bounds on the number of states of the automata. Section 3 then describes the implementation issues, the conversion of the satisfiability problem into a model checking problem, and construction of a circuit corresponding to the automaton. Section 4 provides our experimental results and comparisons with other tools. Finally, Section 5 concludes the paper with the discussion of experimental results and the future work.

2 Presburger Arithmetic

Definition 1. We define Presburger arithmetic to be the first-order theory over atomic formulas of the form

$$\sum_{i=1}^{n} a_i x_i \sim c, \tag{1}$$

where a_i and c are integer constants, x_i's are variables ranging over integers, and $\sim$ is an operator from $\{=, \neq, <, \leq, >, \geq\}$. The semantics of these operators are the usual ones.

In the rest of the paper we restrict ourselves to only *quantifier-free* fragment of Presburger arithmetic.

A *formula* f is either an atomic formula (1), or is constructed from formulas f_1 and f_2 recursively as follows:

$$f ::= \neg f_1 \mid f_1 \wedge f_2 \mid f_1 \vee f_2.$$

Throughout the paper we use the following typographic conventions.

Notation 1. We reserve boldface letters, e.g. $\mathbf{b}$, to represent column vectors and $\mathbf{b}^T$ to represent row vectors. The term *vector* shall always refer to a column vector unless specified otherwise. In this notation, $\mathbf{x}$ represents the vector of variables of the atomic formula:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

and $\mathbf{b}$ represents n-bit boolean column vectors. A row vector of coefficients in an atomic formula is denoted by $\mathbf{a}^T$:

$$\mathbf{a}^T = (a_1,\, a_2,\, \ldots,\, a_n).$$

In particular, an atomic formula in the vector notation is written as follows:

$$f \equiv \mathbf{a}^T \cdot \mathbf{x} \sim c,$$

where $\mathbf{a}^T \cdot \mathbf{x}$ is the scalar product of the two vectors $\mathbf{a}^T$ and $\mathbf{x}$.

We give the formal semantics of the quantifier-free Presburger arithmetic in terms of the sets of solutions. A *variable assignment* for a formula ϕ (not necessarily atomic) with n free variables is an n-vector of integers $\mathbf{w}$. An atomic formula f under a particular assignment $\mathbf{w}$ can be easily determined to be true or false by evaluating the expression $\mathbf{a}^T \cdot \mathbf{w} \sim c$.

A *solution* is a variable assignment $\mathbf{w}$ which makes the formula ϕ true. We denote the set of all solutions of ϕ by $\mathrm{Sol}(\phi)$, which is defined recursively as follows:

- if ϕ is atomic, then $\mathrm{Sol}(\phi) = \{\mathbf{w} \in \mathbb{Z}^n \mid \mathbf{a}^T \cdot \mathbf{w} \sim c\}$;
- if $\phi \equiv \neg\phi_1$, then $\mathrm{Sol}(\phi) = \mathbb{Z}^n - \mathrm{Sol}(\phi_1)$;
- if $\phi \equiv \phi_1 \wedge \phi_2$, then $\mathrm{Sol}(\phi) = \mathrm{Sol}(\phi_1) \cap \mathrm{Sol}(\phi_2)$;
- if $\phi \equiv \phi_1 \vee \phi_2$, then $\mathrm{Sol}(\phi) = \mathrm{Sol}(\phi_1) \cup \mathrm{Sol}(\phi_2)$.

To simplify the definitions, we assume that all atomic formulas of ϕ always contain the same set of variables. If this is not true and some variables are missing in one of the atomic formulas, then these variables can be added with zero coefficients.

2.1 Idea Behind the Automaton

The idea behind the automata-based approach is to construct a *deterministic finite-state automaton* (DFA) A_ϕ for a quantifier-free Presburger formula ϕ such that the language of this automaton $L(A_\phi)$ corresponds to the set of all solutions of ϕ. When such an

automaton is constructed, the satisfiability problem for ϕ is effectively reduced to the *emptiness problem* of the automaton, that is, checking that $L(A_\phi) \neq \emptyset$.

If a formula is not atomic, then the corresponding DFA can be constructed from the DFAs for the subformulas using the complement, intersection, and union operations on the automata. Therefore, to complete our construction of A_ϕ for an arbitrary quantifier-free Presburger formula ϕ it is sufficient to construct DFAs for each of the atomic formulas of ϕ.

Throughout this section we fix a particular atomic Presburger formula f:

$$f \equiv \mathbf{a}^T \cdot \mathbf{x} \sim c.$$

Recall that a variable assignment is an n-vector of integers $\mathbf{w}$. Each integer can be represented in the binary format in 2's complement, so a solution vector can be represented by a vector of binary strings.

We can now look at this representation of a variable assignment $\mathbf{w}$ as a *binary matrix* where each row, or *track*, represents an integer for the corresponding variable, and each i^{th} column represents the vector of the i^{th} bits of all the components of $\mathbf{w}$. Alternatively, this matrix can be seen as a *string of its columns*, a string over the alphabet $\Sigma = \mathbb{B}^n$, where $\mathbb{B} = \{0, 1\}$.

The set of all strings that together represent all the solutions of a formula f form a *language* L_f over the alphabet Σ. Our problem is now reduced to building a DFA for the atomic formula f that accepts exactly the language L_f.

Intuitively, the automaton A_f must read a string π, extract the corresponding variable assignment $\mathbf{w}$ from it, instantiate it into the formula f, and check that the value of the left hand side (LHS) is indeed related to the right hand side (RHS) constant as the relation $\sim$ prescribes. If it does, the string is accepted, otherwise rejected. Since the RHS constant and the relation $\sim$ are fixed in f, the value of the LHS of f solely determines whether the input string π should be accepted or not.

Assume that the automaton A_f reads a string from left to right. If the value of the LHS of f is l after reading the string π, then after appending one more "letter" $\mathbf{b} \in \mathbb{B}^n$ to π on the right, the LHS value changes to $l' = 2l + \mathbf{a}^T \cdot \mathbf{b}$. Notice that only the original value of the LHS l and the new "letter" $\mathbf{b}$ are needed to compute the new value of the LHS l' for the resulting string. This property directly corresponds to the property of the transition relation of an automaton, namely, that the next state is solely determined by the current state and the next input letter.

Following the above intuition, we can define an automaton A_f as follows. The states of A_f are integers representing the values of the LHS of f; the input alphabet is $\Sigma = \mathbb{B}^n$; and on an input $\mathbf{b} \in \Sigma$ the automaton transitions from a state l to $l' = 2l + \mathbf{a}^T \cdot \mathbf{b}$. The set of accepting states are those states l that satisfy $l \sim c$. Special care has to be taken of the initial state $s_{\text{initial}} \notin \mathbb{Z}$. First, we interpret the empty string as a vector of 0's. Thus, the value of the left hand side in the initial state must be equal to 0. The first "letter" read by A_f is the vector of sign bits, and, according to the 2's complement interpretation, the value of the LHS in the next state after s_{initial} must be $l = -\mathbf{a}^T \cdot \mathbf{b}$.

Notice that this automaton is not finite, since we have explicitly defined the set of states to be integers. Later we examine the structure of this infinite automaton and show how to trim the state space to a finite subset and obtain an equivalent DFA, similar to the one in Figure 1.

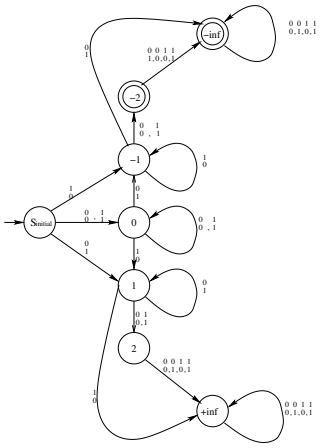

Fig. 1. Example of an automaton for an atomic Presburger formula $x - y \leq -2$.

2.2 Formal Description of the Automaton

An (infinite-state) automaton corresponding to an atomic Presburger formula f is defined as follows:

$$A_f = (S,\, \mathbb{B}^n,\, \delta,\, s_{\text{initial}},\, S_{\text{acc}}),$$

where

- $S = \mathbb{Z} \cup \{s_{\text{initial}}\}$ is the set of states, $\mathbb{Z}$ is the set of integers and $s_{\text{initial}} \notin \mathbb{Z}$;
- s_{initial} is the start state;
- $\mathbb{B}^n$ is the alphabet, which is the set of n-bit vectors, $\mathbb{B} = \{0,\, 1\}$;
- The transition function $\delta : S \times \mathbb{B}^n \to S$ is defined as follows:

$$\begin{aligned} \delta(s_{\text{initial}},\, \mathbf{b}) &= -\mathbf{a}^T \cdot \mathbf{b} \\ \delta(l,\, \mathbf{b}) &= 2l + \mathbf{a}^T \cdot \mathbf{b} \end{aligned}$$

  where $l \in \mathbb{Z}$ is a non-initial state.
- The set of accepting states

$$S_{\text{acc}} = \{l \in \mathbb{Z} \mid l \sim c\} \cup \begin{cases} \{s_{\text{initial}}\} & \text{if } \mathbf{a}^T \cdot \mathbf{0} \sim c \\ \emptyset & \text{otherwise.} \end{cases}$$

In the rest of this section we show how this infinite automaton can be converted into an equivalent finite-state automaton. Intuitively, there is a certain finite range of values of the LHS of f such that if A_f transitions outside of this range, it starts *diverging*, or "moving away" from this range, and is guaranteed to stay outside of this range and on the same side of it (i.e. diverging to $+\infty$ or $-\infty$). We show that all of the states outside of the range can be collapsed into only two states (representing $+\infty$ and $-\infty$ respectively), and that those states can be meaningfully labeled as accepting or rejecting without affecting the language of the original automaton A_f.

Definition 2. For a vector of LHS coefficients $\mathbf{a}^T = (a_1, \ldots, a_n)$ define

$$||\mathbf{a}^T||_- = \sum_{\{i|a_i<0\}} |a_i|$$

$$||\mathbf{a}^T||_+ = \sum_{\{i|a_i>0\}} |a_i|$$

Notice that both $||\mathbf{a}^T||_-$ and $||\mathbf{a}^T||_+$ are non-negative. Let $\mathbf{b}$ denote an n-bit binary vector, that is, $\mathbf{b} \in \mathbb{B}^n$.

Observe that $-\mathbf{a}^T \cdot \mathbf{b} \leq ||\mathbf{a}^T||_-$ for any value of $\mathbf{b}$, since the expression $-\mathbf{a}^T \cdot \mathbf{b}$ can be rewritten as

$$-\mathbf{a}^T \cdot \mathbf{b} = \left(\sum_{\{j \mid a_j<0\}} |a_j| b_j \right) - \left(\sum_{\{i \mid a_i>0\}} |a_i| b_i \right).$$

Therefore, the largest positive value of $-\mathbf{a}^T \cdot \mathbf{b}$ can be obtained by setting b_i to 0 whenever $a_i > 0$, and setting b_j to 1 when $a_j < 0$, in which case $-\mathbf{a}^T \cdot \mathbf{b} = ||\mathbf{a}^T||_-$. It is clear that any other assignment to $\mathbf{b}$ can only make $-\mathbf{a}^T \cdot \mathbf{b}$ smaller. Similarly, $\mathbf{a}^T \cdot \mathbf{b} \leq ||\mathbf{a}^T||_+$.

Lemma 3. *Given an atomic Presburger formula $\mathbf{a}^T \cdot \mathbf{x} \sim c$, a corresponding automaton A_f as defined is Section 2.2, and a current state of the automaton $l \in \mathbb{Z}$, the following two claims hold:*

1. *If $l > ||\mathbf{a}^T||_-$, then any next state l' will satisfy $l' > l$.*
2. *If $l < -||\mathbf{a}^T||_+$, then any next state l' will satisfy $l' < l$.*

Proof. **The upper bound (claim 1).** Assume that $l > ||\mathbf{a}^T||_-$ for some state $l \in \mathbb{Z}$. Then the next state l' satisfies the following:

$$\begin{aligned} l' &= 2l + \mathbf{a}^T \cdot \mathbf{b} \\ &\geq 2l - ||\mathbf{a}^T||_- \\ &> 2l - l = l. \end{aligned}$$

The lower bound (claim 2) is similar to the proof of claim 1.

□

We now discuss bounds on the states of the automata based on Lemma 3. From this lemma it is easy to see that once the automaton reaches a state outside of

$$\left[\min(-||\mathbf{a}^T||_+, c),\ \max(||\mathbf{a}^T||_-, c)\right],$$

it is guaranteed to stay outside of this range and on the same side of it. That is, if it reaches a state $l < \min(-||\mathbf{a}^T||_+, c)$, then $l' < \min(-||\mathbf{a}^T||_+, c)$ for any subsequent state l' that it can reach from l. If the relation $\sim$ in f is an equality, then $l = c$ is guaranteed to be false from the moment A_f transitions to l onward. Similarly, it will be false forever when $\sim$ is $\geq$ or $>$; however it will always be true for $<$ and $\leq$ relations.

In any case, either all of the states l of the automaton A_f below $\min(-||\mathbf{a}^T||_+, c)$ are accepting, or all of them are rejecting. Since the automaton will never leave this set of states, it will either always accept any further inputs or always reject. Therefore, replacing all states below $\min(-||\mathbf{a}^T||_+, c)$ with one single state $s_{-\infty}$ with a self-loop transition for all inputs and marking this state appropriately as accepting or rejecting will result in an automaton equivalent to the original A_f. Exactly the same line of reasoning applies to the states $l > \max(||\mathbf{a}^T||_-, c)$, and they all can be replaced by just one state $s_{+\infty}$ with a self-loop for all inputs.

Formally, the new *finite* automaton has the set of states

$$S = \left[\min(-||\mathbf{a}^T||_+, c),\ \max(||\mathbf{a}^T||_-, c)\right] \cup \{s_{\text{initial}},\ s_{-\infty},\ s_{+\infty}\}.$$

Transitions within the range coincide with the transitions of the original (infinite) automaton A_f. If in the original automaton $l' = \delta(l, \mathbf{b})$ for some state l and input $\mathbf{b}$, and $l' > \max(||\mathbf{a}^T||_-, c)$, then in the new automaton the corresponding next state is $\delta'(l, \mathbf{b}) = s_{+\infty}$, and subsequently, $\delta'(s_{+\infty}, \mathbf{b}) = s_{+\infty}$ for any input $\mathbf{b}$. Similarly, if the next state $l' < \min(-||\mathbf{a}^T||_+, c)$, then the new next state is $s_{-\infty}$, and the automaton remains in $s_{-\infty}$ forever:

$$\begin{aligned}
\delta'(s_{\text{initial}}, \mathbf{b}) &= -\mathbf{a}^T \cdot \mathbf{b} \\
\delta'(s_{+\infty}, \mathbf{b}) &= s_{+\infty} \\
\delta'(s_{-\infty}, \mathbf{b}) &= s_{-\infty} \\
\delta'(l, \mathbf{b}) &= \begin{cases} s_{+\infty}, & \text{if } 2l + \mathbf{a}^T \cdot \mathbf{b} > \max(||\mathbf{a}^T||_-, c) \\ s_{-\infty}, & \text{if } 2l + \mathbf{a}^T \cdot \mathbf{b} < \min(-||\mathbf{a}^T||_+, c) \\ 2l + \mathbf{a}^T \cdot \mathbf{b}, & \text{otherwise.} \end{cases}
\end{aligned}$$

The accepting states within the range are those that satisfy the $\sim$ relation. The new "divergence" states are labeled accepting if the $\sim$ relation holds for some representative state. For instance, for a formula

$$\mathbf{a}^T \cdot \mathbf{x} < c$$

the state $s_{-\infty}$ is accepting, and $s_{+\infty}$ is rejecting. Finally, the initial state s_{initial} is accepting if and only if it is accepting in the original infinite automaton.

We can use the bounds from Lemma 3 to repeat the analysis from [19] for the number of states of the automaton and obtain new bounds tighter by a factor of 2. Since we have to know the bounds in advance when constructing an SMV model, this saves one bit of state for every atomic formula. Asymptotically, of course, our new bounds stay the same as in [19].

3 Implementation

In the previous section we have shown a mathematical construction of a deterministic finite-state automaton corresponding to a quantifier-free Presburger formula f. In practice, building such an automaton explicitly is very inefficient, since the number of states is proportional to the value of the coefficients in $\mathbf{a}^T$ and the right hand side constant c and, most importantly, the number of transitions from each state is exponential (2^n) in the number of variables in f.

Instead, we use an existing *symbolic model checker* SMV [13] as a means to build the symbolic representation of the automaton and check its language for emptiness. Symbolic model checking expresses a design as a finite-state automaton, and then properties of this design are checked by traversing the states of the automaton. In the past decade, there has been a lot of research in boosting the performance of model checkers. Most notable breakthrough was in early 90s when binary decision diagrams [4] (BDDs) were successfully used in model checking [13], pushing the tractable size of an automaton to as many as 10^{20} states and beyond [6].

Therefore, it is only natural to try to utilize such powerful and well-developed techniques of handling finite-state automata in checking the satisfiability of Presburger formulas. The obvious advantages of this approach is that the state-of-the-art verification engines such as SMV are readily available, and the only remaining task is to transform the emptiness problem for an automaton into a model checking problem efficiently. In addition, with SMV we exploit the efficient BDD representation for both states and transitions of the automata, whereas in the other automata-based approaches like MONA or LASH the states are represented explicitly.

We have performed all of our experiments with the CMU version of SMV model checker. Although the SMV language allows us to express the automaton and its transitions directly in terms of arithmetic expressions, the cost of evaluating these expressions in SMV is prohibitively high.

Internally, SMV represents all the state variables as vectors of boolean variables. Similarly, the representation of the transition relation is a function[2] that takes boolean vectors of the current state variables and the inputs and returns new boolean vectors for the state variables in the next state.

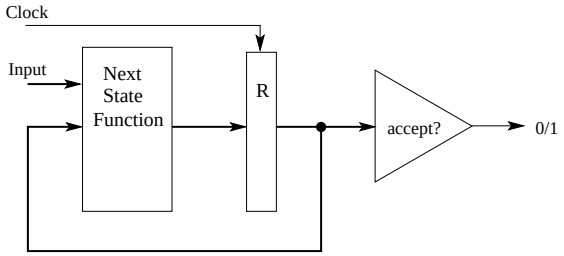

Fig. 2. Circuit implementing a finite-state automaton.

Effectively, SMV builds an equivalent of a *sequential digital circuit* operating on boolean signals, as shown in Figure 2. The current state of the automaton is stored in the *register* R. The next state is computed by a combinational circuit from the value of the current state and the new inputs, and the result is latched back into the register R at the next clock cycle. A special *tester circuit* checks whether the current state is

[2] Strictly speaking, SMV constructs a *transition relation* which does not have to be a function, but here it is indeed a function, so this distinction is not important.

accepting, and if it is, the sequence of inputs read so far (or the *string* in our original terminology) is accepted by the automaton (and represents a solution to f).

The property that we check is that the output of the circuit never becomes 1 for any sequence of inputs. In the logical specification language of SMV, this is written as $\mathbf{AG}(\mathrm{output} \neq 1)$. If this property is true, then the language of the automaton is empty, and the original formula f is unsatisfiable. If this property is violated, SMV generates a *counterexample trace* which is a sequence of transitions leading to an accepting state. This trace represents a *satisfying assignment* to the formula f.

The translation of the arithmetic expressions to such a boolean circuit is the primary bottleneck in SMV. Hence, providing the circuit *explicitly* greatly speeds up the process of building the transition relation.

A relatively simple Perl script generates such a circuit and the property very efficiently and transforms it into a SMV description. The structure of the resulting SMV code follows very closely the mathematical definition of the automaton, but all the state variables are explicitly represented by several boolean variables, and all the arithmetic operations are converted into combinational circuits (or, equivalently, boolean expressions). In particular, ripple-carry adders are used for addition, "shift-and-add" circuits implement multiplication by a constant, and comparators implement equality and inequality relations in the tester circuit.

4 Experimental Results

Since the satisfiability problem for quantifier-free Presburger arithmetic is NP-complete, the hope that it has an efficient general purpose decision procedure is quite thin. Therefore, for practical purposes, it is more important to collect several different methods and evaluate their performance on different classes of formulas. When strengths and weaknesses of each of the approaches and tools are identified, it is easier to pick the best one for solving concrete problems that arise in practice.

The primary purpose of our experiments is to study the performance of automata-based and ILP-based methods and their variations depending on different parameters of Presburger formulas. The tools and approaches that we picked are the following:

- Automata-based tools:
  - Our approach using the SMV model checker (we refer to it as "SMV");
  - LASH [19], a direct implementation of the automata-based approach dedicated to Presburger arithmetic;
  - MONA [10], an automata-based solver for WS1S and a general-purpose automata library.
- Approaches based on Integer Linear Programming (ILP):
  - LP_SOLVE, simplex-based open source tool with branch-and-bound for integer constraints;
  - CPLEX, one of the best commercial simplex-based LP solvers;
  - OMEGA [16], a tool based on Fourier-Motzkin algorithm [18].

The benchmarks consist of many randomly generated relatively small quantifier-free Presburger formulas. The examples have three main parameters: the number of variables, the number of atomic formulas (the resulting formula is a conjunction of atomic formulas), and the maximum value of the coefficients. For each set of parameters we generate 5 random formulas and run this same set of examples through each of the tools.

The results of the comparisons appear in Figures 3, 4, and 5 as plots showing how execution time of each automata-based tool depends on some particular parameter with other parameters fixed, and the success rate of all the tools for the same parameters. Each point in the run-time graphs represents a *successful run* of an experiment in a particular tool. That is, if a certain tool has fewer points in a certain range, then it means it failed more often in this range (ran out of memory or time, hit a fatal error, etc.). The ILP tools either complete an example within a small fraction of a second, or fail. Therefore the run-time is not as informative for ILP tools as the number of completed examples, and hence, only the success rates for those are shown.

In the case of MONA, the only readily available input language is WS1S, and we have found that translating Presburger formulas into WS1S is extremely inefficient. Even rather simple examples which SMV and LASH solve in no time take significant time in MONA. Due to this inefficient translation, the comparison of MONA with other approaches is not quite fair. Therefore, it is omitted from the graphs and will not be considered in our discussion further.

LASH and SMV both have obvious strengths and weaknesses that can be easily characterized. SMV suffers the most from the number of atomic formulas, as can be seen from Figure 3 where the run-time is plotted as a function of the number of atomic formulas. The largest number of formulas it could handle in this batch is 11, whereas the other tools including LASH finished most of the experiments with up to 20 atomic formulas. This suggests that the implemention of the parallel composition of automata for atomic formulas in SMV is suboptimal. LASH apparently has a better way of composing automata.

Varying the number of variables (Figure 4) makes SMV and LASH look very much alike. Both tools can complete all of the experiments, and the run-time grows approximately exponentially with the number of variables and at the same rate in both tools. This suggests that the BDD-like structure for the transitions in LASH indeed behaves very similarly to BDDs in SMV.

However, since the number of states in the automata are proportional to the values of the coefficients, LASH cannot complete any of the experiments with coefficients larger than 4096 and fails on many experiments even with smaller values. SMV, on the other hand, can handle as large coefficients as 2^{30} with only a moderate increase of the run-time and the failure rate. We attribute this behavior to the fact that in SMV both the states and the transitions of the automata are represented with BDDs, while in LASH (and all the other available automata-based tools) the states are always represented explicitly.

Finally, we have to say a few words about the ILP based methods. First of all, these methods are greatly superior to the automata-based in general, and they do not exhibit any noticeable increase in run-time when the number of variables or the number

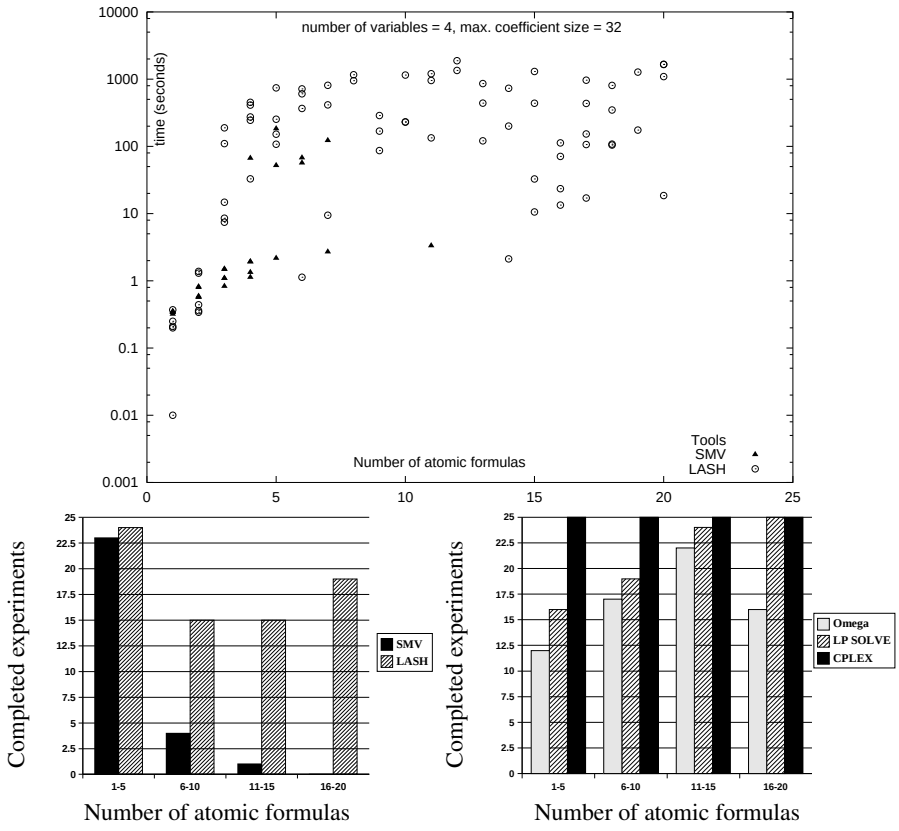

Fig. 3. Run-time and the number of completed experiments depending on the number of atomic formulas in each test case.

of formulas increase. The only limiting factor for ILPs are the values of the coefficients, which cause many failures and overflows starting at about 10^7, especially in LP_SOLVE. Although all of the successful runs of the ILP-based tools are well under a fraction of a second, there are also many failures due to a non-terminating branch-and-bound search, overflow exceptions, and program errors. OMEGA is especially notorious for segmentation faults, and its failure rate greatly increases when the values of the coefficients approach the limit of the machine-native integer or float representation.

Despite overall superiority of the ILP-based methods over the automata-based ones, there are a few cases where the ILP methods fail while the automata-based methods work rather efficiently. The most interesting class of such examples can be characterized as follows. The formula must have a solution in real numbers, but the integer solutions either do not exist or they are rather sparse in the *feasibility set* (the set of real solutions) of the formula. Additionally, the direct implementation of the branch-and-bound method is incomplete when the feasibility set is unbounded, since there are infinitely

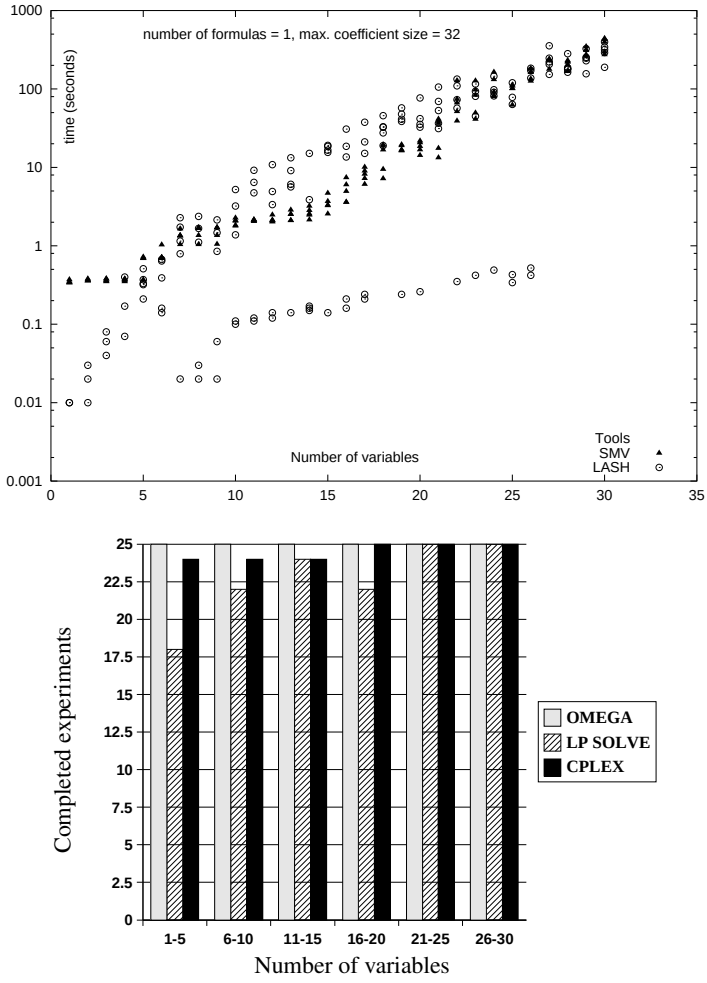

Fig. 4. Run-time and the number of completed experiments depending on the number of variables in a single atomic formula. SMV and LASH finish all of the experiments, hence there is no bar chart for those.

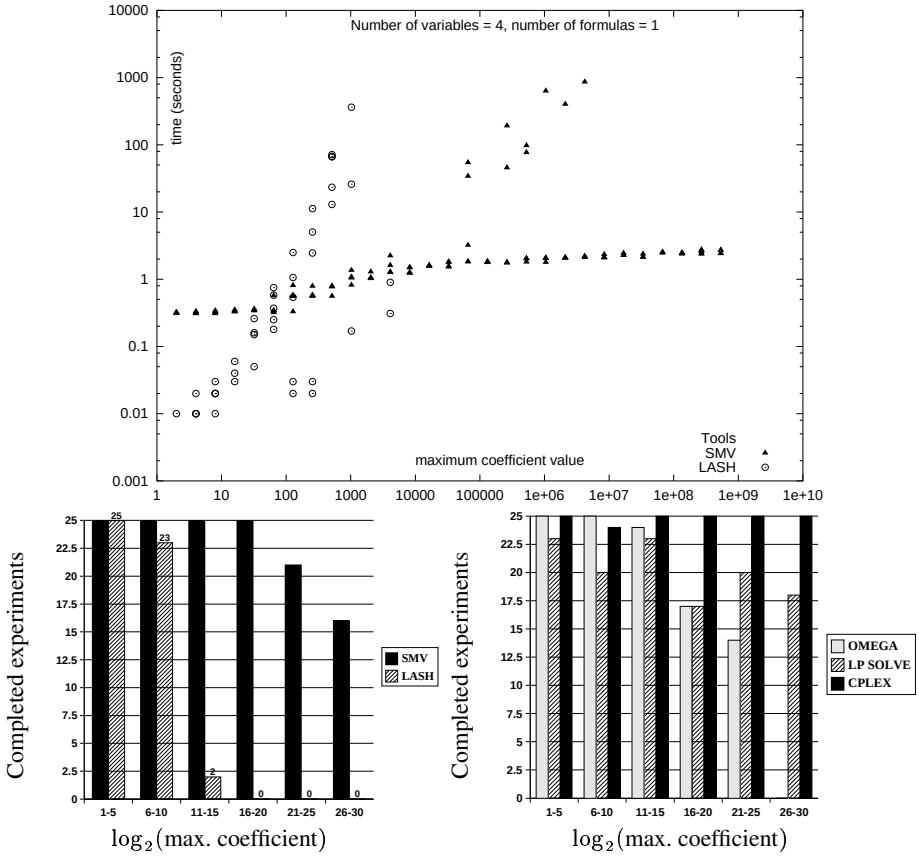

Fig. 5. Run-time and the number of completed examples depending on the (maximum) values of the coefficients in a single atomic formula.

many integer points that have to be checked. This claim still holds to some extent even in the heuristic-rich top quality commercial tools such as CPLEX, and we have observed their divergence on a few examples that are trivial even for the automata-based techniques.

The OMEGA approach stands out from the rest of ILP tools since it is based on the Fourier-Motzkin method which is complete for integer linear constraints. Unfortunately, the only readily available implementation of this method is very unstable.

Another common weakness of all of the ILP-based approaches is the limit of the coefficient and solution values due to the rounding errors of native computer arithmetic. It is quite easy to construct an example with large integer coefficients for which CPLEX returns a plainly wrong answer. Large coefficients can be extremely useful in hardware verification when operations on long bit-vectors are translated into Presburger arithmetic.

We conjecture that the efficiency of the ILP methods highly depends on the use of computer arithmetic, and the only fair comparison with automata-based methods can be done if the ILP tools use arbitrary precision arithmetic.

5 Conclusion

Efficient decision procedures for Presburger arithmetic are key to solving many formal verification problems. We have developed a decision procedure based on the idea of converting the satisfiability problem into a model checking problem. Experimental comparisons show that our method can be more efficient than other automata-based methods like LASH and MONA, particularly for formulas with large coefficients. In our approach we use BDDs both for the states and the transitions of the automata while LASH and MONA use BDDs or similar structures only for the transitions. As an additional theoretical result, we provide tighter bounds for the number of states of the automata. This makes our automaton construction in SMV even more efficient.

Another advantage of our approach is that converting the satisfiability problem into model checking problem requires very little implementation effort. We exploit the existing SMV model checker as a back-end which employs a very efficient BDD package. Therefore, the only effort required from us is the translation of a Presburger formula into the SMV input language.

In addition, we compare various automata and ILP-based approaches on a suite of parameterized randomly generated Presburger formulas. For every approach we identify classes of Presburger formulas for which it either performs very poorly or very efficiently. For instance, we found that the ILP-based tools are more likely to fail on examples with unbounded but sparse solution sets and cannot handle large coefficients due to the use of native machine arithmetic. The automata-based tools are not as sensitive to these parameters. On the other hand, ILP-based approaches scale much better on the number of variables and atomic formulas. We also believe that the ILP tools have an unfair advantage over the automata methods due to the use of native arithmetic. However, until further experiments are done with an ILP tool with support for arbitrarily large integers we cannot tell how much difference it makes.

Within the automata-based approaches SMV scales better with the coefficients' size, but displays poorer performance for large number of atomic formulas when compared to LASH. Both perform equally well as the number of variables is varied.

The reason the other tools do not use BDDs for the states is because they perform quantifier elimination by manipulating the automata directly. Namely, each quantifier alternation requires projection and determinization of the automaton. The use of BDDs for the states can make the implementation of the determinization step particularly hard. This difference is one of the reasons for the relative efficiency of our approach.

The extension of our approach to full Presburger arithmetic can be done by combining it with the traditional quantifier elimination method [12]. This method introduces a new type of atomic formulas with the divisibility operator: $\mathbf{a}^T \cdot \mathbf{x} \mid c$, and our automaton construction can be easily extended to handle it.

We also believe that our approach may prove useful to other theories and logics which use automata based decision procedures.

References

1. Tod Amon, Gaetano Borriello, Taokuan Hu, and Jiwen Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Design Automation Conference*, pages 226–231, 1997.
2. Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Colloquium on Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer Verlag, 1996.
3. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *IEEE VLSI Design'01 & Asia and South Pacifi c Design Automation Conference, Bangalore*, pages 741–746, 2002.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
5. J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundladen der Mathematik*, 6:66–92, 1960.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
7. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
8. D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.
9. George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.
10. Jacob Elgaard, Nils Klarlund, and Anders Mller. Mona 1.x: new techniques for ws1s and ws2s. In *Computer Aided Verifi cation, CAV '98, Proceedings*, volume 1427 of *LNCS*. Springer Verlag, 1998.
11. P. Johannsen and R. Drechsler. Formal verification on the RT level computing one-to-one design abstractions by signal width reduction. In *IFIP International Conference on Very Large Scale Integration (VLSI'01), Montpellier, 2001*, pages 127–132, 2001.
12. G. Kreisel and J. Krivine. Elements of mathematical logic, 1967.

13. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
14. Derek C. Oppen. A $2^{2^{2pn}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, June 1978.
15. M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101, 395, Warsaw, 1927.
16. William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.
17. T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verifi cation*, volume 1427, pages 280–292. Springer-Verlag, 1998.
18. H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory (A)*, 21:118–123, 1976.
19. Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.