A Comparison of Presburger Engines for EFSM Reachability

Thomas R. Shiple[1] James H. Kukula[2] Rajeev K. Ranjan[1]

[1] Synopsys, Inc., Mountain View, CA. {shiple,rajeevr}@synopsys.com
[2] Synopsys, Inc., Beaverton, OR. kukula@synopsys.com

Abstract. Implicit state enumeration for extended finite state machines relies on a decision procedure for Presburger arithmetic. We compare the performance of two Presburger packages, the automata-based Shasta package and the polyhedra-based Omega package. While the raw speed of each of these two packages can be superior to the other by a factor of 50 or more, we found the asymptotic performance of Shasta to be equal or superior to that of Omega for the experiments we performed.

1 Introduction

Peano arithmetic, the theory of arithmetic with multiplication and addition, is undecidable. However, decision procedures do exist for the subset of arithmetic, known as Presburger arithmetic, that excludes multiplication [13]. Presburger formulas are built up from natural number constants, natural number variables, addition, equality, inequality, and the first order logical connectives. An example of such a formula is[1]

$$\exists x(y = 2x + 1).$$

Even though the best known procedure for deciding Presburger arithmetic is triply exponential in the length of the formula [16], several practical applications for Presburger arithmetic have been found. Pugh [17] uses Presburger arithmetic for data dependence analysis in optimizing compilers. Amon *et al.* [1] use Presburger arithmetic to perform symbolic verification of timing diagrams. Another application, and the one on which we will be focusing, is reachability analysis of extended finite state machines (EFSMs) [3,8,9,12].

An EFSM is a system with a finite state controller interacting with an integer datapath of unbounded width [9]. Each transition of the controller has a gating predicate over the integer variables, and an update function specifying the new values of the integer variables when the transition is taken. Figure 1 depicts a simple EFSM with five control states, seven input variables $(r, i_{a_x}, i_{a_y}, i_{b_x}, i_{b_y}, i_{d_x}, i_{d_y})$, seven data variables $(a_x, a_y, b_x, b_y, d_x, d_y, i)$, and ten transitions. This machine reads data and then checks a series of inequalities that determines whether the variable i should be assigned a 0 or 1 value.

[1] $2x$ is an abbreviation for $x + x$.

If the gating predicates and update functions of an EFSM are definable in Presburger arithmetic, then the entire transition relation of the EFSM can be represented as a single Presburger formula. If the set of initial states is also Presburger definable, then BFS-based implicit state enumeration can be performed completely within Presburger arithmetic. Thus, Presburger arithmetic provides an elegant framework for performing state reachability of EFSMs.

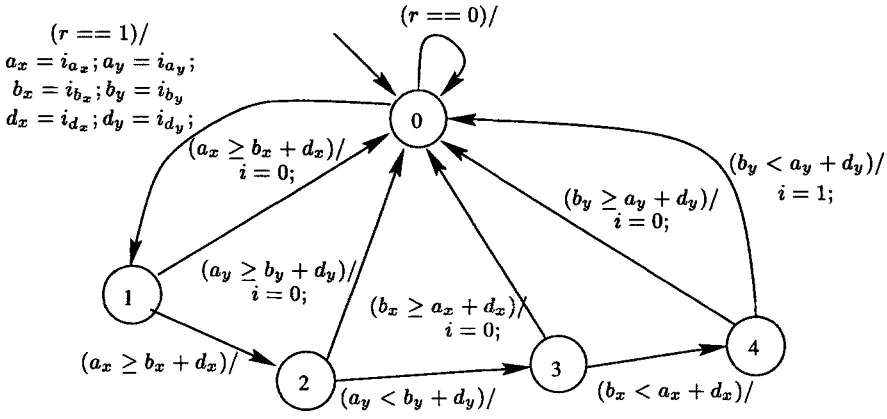

Fig. 1. An EFSM.

Two basic approaches have emerged for representing, manipulating, and checking the satisfiability of Presburger formulas: automata-based and polyhedra-based. In the automata-based approach, the naturals are encoded as bit strings using base 2 encoding [4, 6, 19]. For a Presburger formula defined over k variables, a technique has been developed to directly translate the formula into a deterministic, finite state automaton (DFA) that accepts a k-tuple of bit strings if and only if the k-tuple is a solution to the given formula [4]. Since minimum state DFAs are unique, automata provide a canonical form for Presburger formulas. In the polyhedra-based approach, Fourier-Motzkin variable elimination is used to eliminate the quantifiers from a Presburger formula [13, 17]. The result is a union of convex polyhedra that is typically represented by a set of matrices; this representation is not canonical. A useful analogy can be made to data representations in the Boolean domain: automata are like binary decision diagrams (BDDs), and polyhedra are like sums of products (SOPs).

To the best of our knowledge, a direct experimental comparison of the performance of these two basic approaches has never been made. The contribution of this work is to perform such a comparison. For the polyhedra approach, we use the Omega package of Pugh *et al.* [15, 17]. For the automata approach, we developed the Shasta package, which incorporates the procedure of Boudet and Comon [4] for translating linear equalities and inequalities to automata, and also uses the automaton data structure of Henriksen *et al.* [14].

The context for our comparison is state reachability for EFSMs. It is not clear *a priori* which approach would be better. In the Boolean domain, experience shows that BDDs are generally superior to SOPs, but there are cases where SOPs are exponentially more compact than BDDs [11]. In the final analysis, our experiments show that while the raw speed of each Presburger engine can be superior to the other by a factor of 50 or more, the asymptotic performance of Shasta is equal or superior to that of Omega.

The remainder of the paper is organized as follows. Section 2 presents more detail on the automata approach for solving Presburger arithmetic, and Section 3 does the same for the polyhedra approach. Section 4 describes the experimental setup and analyzes the experimental results.

2 Automata Approach for Solving Presburger Arithmetic

Automata can be used as a data structure to represent Presburger formulas, or more precisely, the set of solutions to Presburger formulas. In this section, we review the general ideas behind this concept, and discuss some specifics of the Shasta automata package used in the experiments.

The first step to consider in representing Presburger formulas by automata is the encoding of natural numbers. For this, a base 2 encoding is used, with least significant bit first, and arbitrary padding with zeroes on the end. Thus, both 011 and 01100 represent the number 6. A tuple of naturals is represented by simply stacking equal length representations of the elements. Thus, the tuple $(x_1, x_2, x_3) = (4, 7, 11)$ can be represented by a string of bit vectors:

$$\begin{array}{lc} \text{string} \longrightarrow & \\ x_1\!: 0\;0\;1\;0 & \text{bit} \\ x_2\!: 1\;1\;1\;0 & \text{vector} \\ x_3\!: 1\;1\;0\;1 & \downarrow \end{array}$$

An automaton representing a Presburger formula over k variables reads a bit vector of height k at each step, consuming the least significant bit of each variable in the first step, the next least significant bit in the second step, and so on.

The atomic formulas of Presburger arithmetic are linear equalities and inequalities. Büchi indirectly showed how these formulas can be represented by automata by demonstrating how they can be embedded in the logic WS1S [6]. Recently, Boudet and Comon developed a direct method for translating an atomic formula into an automaton; the Shasta package uses this algorithm. Figure 2 shows the automaton that recognizes the natural number tuples satisfying the linear equality $x_1 + x_2 = x_3$. To illustrate its operation, for the input tuple (4,7,11), the automaton starts at the initial state and reads the first bit vector, 011, which leads the automaton back to the initial state. After reading all the bit vectors, the automaton will be in the accepting state, reflecting that $4 + 7 = 11$.

Presburger formulas are constructed by combining atomic formulas using the first order logical connectives. These connectives are handled by standard automata operations: logical conjunction translates to automata intersection, logical negation to automata complementation, and existential quantification to automata projection. Quantification deserves a closer look. Consider the formula $\exists x_2(x_1 + x_2 = x_3)$; this defines the relation $x_1 \leq x_3$. The automaton for this formula can be derived by simply "erasing" the second component of each transition label of the automaton in Figure 2, yielding a nondeterministic automaton. To return to a canonical form, this automaton would

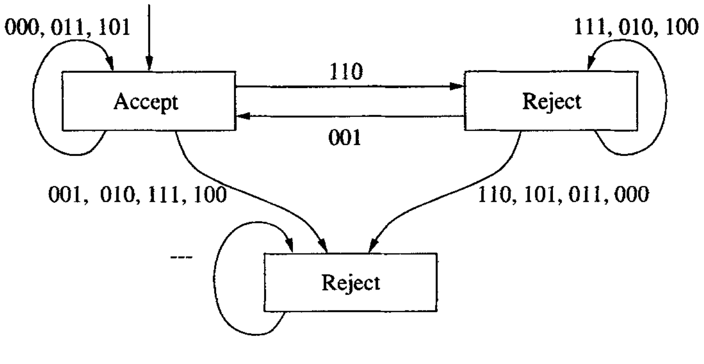

Fig. 2. The automaton representing $x_1 + x_2 = x_3$.

need to be determinized and minimized (Shasta automatically applies state minimization after every operation). Given a minimized automaton, satisfiability can be checked in constant time.

The Shasta package incorporates the automaton data structure of Henriksen *et al.* [14]. Rather then having 2^k labels annotating the outgoing transitions of each state, a BDD with multiple terminals is used. Specifically, each state of an automaton points to a BDD that determines the next state as a function of the incoming bit vector. The terminal nodes of the BDD are the possible next states, and the BDDs for different states can share common subgraphs. Figure 3 shows the same automaton as Figure 2, with its transitions represented by BDDs.

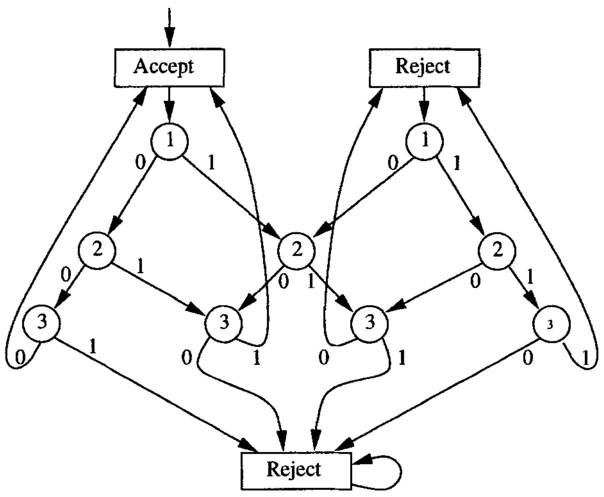

Fig. 3. The automaton representing $x_1 + x_2 = x_3$, with transitions represented by BDDs.

Shasta actually goes slightly beyond Presburger arithmetic by treating Boolean variables specially, rather than just as natural number variables that only take the values 0 and 1. This is done simply by ordering all the Boolean variables first, and reading their values just once. The effect at the data structure level is an automaton rooted by a "pure" BDD, whose terminals are states of the automaton.

Presburger arithmetic is strictly defined over just the naturals, not all the integers. The Shasta package follows this definition. However, it is possible to extend Presburger arithmetic to the integers by encoding each integer as a pair of natural numbers [18].

3 Polyhedra Approach for Solving Presburger Arithmetic

The set of solutions to a Presburger formula can be represented in a sum of products form, whose primitive formulas are linear equalities, inequalities, and congruences:

$$\bigvee_k \left[\bigwedge_j (0 = a_{0jk} + \sum_i a_{ijk} x_i) \wedge \bigwedge_j (0 < b_{0jk} + \sum_i b_{ijk} x_i) \right. \\ \left. \wedge \bigwedge_j (0 \equiv_{d_{jk}} c_{0jk} + \sum_i c_{ijk} x_i) \right] \qquad (1)$$

Here $\equiv_d$ means equivalent modulo d, a_{ijk}, b_{ijk}, c_{ijk}, and d_{jk} are all integer constants, and x_i are integer variables. This formula can be given a geometric interpretation: the conjunction of equalities defines a linear subspace, the conjunction of inequalities defines a convex polyhedron, and the congruences pick out periodic sets.

The essential function required of a Presburger engine is to check the satisfiability of a formula. To check satisfiability in a polyhedra-based approach, a Presburger formula must first be converted to a sum of products, as in Equation 1. The next step is to existentially quantify any free variables. If the resulting formula, involving only constants, is true, then the original formula is satisfiable. Thus, we need to construct, for any Presburger formula, a representation in the form of Equation 1 [13]. As mentioned in Section 2, Presburger formulas are built up from linear equalities and inequalities using conjunction, complementation, and quantification. Linear equalities and inequalities are just trivial instances of the representation in Equation 1. Any conjunction can be simply distributed over the disjunctions to produce a new disjunctive form. Complementation of an entire SOP formula can be converted by De Morgan's rule into combinations of complementations of primitive formulas. The complements of the primitive formulas can be expressed in terms of uncomplemented primitive formulas.

Lastly, we need to see how to existentially quantify a variable in this representation and express the result in the same form. The details of this operation are too complex to provide in this brief treatment, but the main steps can be outlined. First, to quantify a variable x, all the primitive formulas need to be scaled so that x appears with the same coefficient, a, which will be the least common multiple of the coefficients of x in the original primitive formulas. Then ax can be replaced by a new variable y, adding the new term $0 \equiv_a y$ to our formula.

Finally, to eliminate y, if there any equalities that include y then one can just use Gaussian elimination, picking one equality to provide a formula to substitute for y in all its other occurrences. If y does not occur in any equality, then Fourier-Motzkin elimination can be used to check the various inequalities for the existence of a solution. The basic idea is that for every pair of inequalities $f < y$ and $y < g$, the inequality $f < g$ must be satisfied. One also needs to guarantee that the gap between f and g includes some integer that satisfies the various congruence equations. There are a finite number of congruence classes, so the possible solutions can be enumerated. Enumerating the pairs of inequalities and the congruence classes can generate a large number of new inequalities. This potential for combinatorial explosion is what makes Presburger arithmetic complex.

The main challenge in using this polyhedra-based representation is efficiency. The representation is not canonical. Given one representation for the set of solutions to a formula, one can apply various minimization tactics to search for a smaller equivalent representation. These tactics can get very expensive and one cannot tell in advance whether they will succeed in reducing the size of the representation. At the same time very simple tactics can be quite effective.

The Omega package [15, 17] uses sophisticated versions of these techniques to provide a complete set of Presburger arithmetic operations. It offers user control over when, and to what degree, minimization of formulas should be applied. With such polyhedra-based techniques, it is most natural to support positive and negative numbers on an equal footing. Note that the Omega package does not provide any direct support for Boolean variables, unlike the Shasta package.

4 Experimental Results

The purpose of the experiments is to compare the relative performance of the automata-based Shasta engine to the polyhedra-based Omega engine. In this section, we first discuss in more detail the context of the experiments, namely EFSM reachability, then describe the experimental framework and examples used, present the experimental data, and finally draw some conclusions.

4.1 EFSM Reachability

EFSMs differ from FSMs in that some of the input and state variables of an EFSM can be unbounded natural numbers. Nonetheless, the BFS-based implicit state enumeration technique used for FSMs [10] can be carried directly over to the EFSM domain. Let x_i, x_s, x'_s, and x_o represent the sets of natural number variables[2] for the inputs, present states, next states, and outputs, respectively, of an EFSM, and let I and T represent the set of initial states and the monolithic transition relation, respectively. Then the set of states reachable in j or fewer steps is given by:

$$\begin{aligned} R_0(x_s) &= I(x_s) \\ R_j(x'_s) &= R_{j-1}(x'_s) \vee \exists x_s, x_i, x_o(R_{j-1}(x_s) \wedge T(x_i, x_s, x'_s, x_o)) \end{aligned}$$

[2] For this explanation, we do not distinguish Boolean variables.

If I and T are Presburger definable, then R_j is a Presburger formula, and hence the entire calculation can be carried out using a Presburger engine. One major difference between FSM state enumeration and EFSM state enumeration is that the latter is not guaranteed to converge, because EFSMs are infinite state systems. Our reachability algorithm tests for convergence after each step; some of our examples converge, others do not.

4.2 Experimental Framework and Examples

The examples are described in a dialect of Verilog that includes wires carrying unbounded integer values, and arithmetic modules that operate on them. In particular, each example is specified as a multi-level circuit, where the components can be adders, subtractors, multiplexors, comparators, Boolean logic gates, and Boolean and integer valued flip-flops. There is a single clock that drives all the flip-flops.

We incorporated EFSM reachability into the VIS program [5] by making several modifications and additions to VIS. First, we added a generic Presburger engine interface which, at the flip of a runtime switch, can use either the Shasta or Omega engines. This way we perform the same sequence of elementary operations with both engines, ensuring a fair comparison. Second, we modified the front end of VIS to accept the Verilog dialect mentioned above. Third, we wrote a new routine to build the transition relation using a series of generic Presburger engine calls. Specifically, a monolithic transition relation is built by introducing a variable for each internal circuit net, constructing the input/output relation of each circuit component, forming the conjunction of all the component relations, and then existentially quantifying all the internal variables. Finally, we added a new reachability routine that also makes use of generic Presburger calls. Customary BDD techniques, such as early variable quantification and the use of don't cares for minimization, could be applied in the Presburger framework also, but this has not been done.

We developed several small EFSM examples; these are either typical circuits found in DSP, communication protocol, and computer applications, or they are intended to test hypotheses regarding the relative strengths of the two engines. The circuits are briefly characterized in Table 1; *sequential depth* refers to the greatest lower bound on the path length from an initial state to any reachable state. A brief description of each example follows.

- "ticket" is the ticket mutual-exclusion algorithm from [8], with 2 clients. A client can enter the critical section when its local ticket number becomes equal to the last used ticket number, plus one. An extra Boolean input is used to model the interleaving semantics used in [8].
- "perfect" reads a number a and then computes the sum of all the divisors of a (excluding a itself). If the sum equals a, then a is called "perfect".
- "sdiv" is a serial divider. A numerator and denominator are read and saved, and then the denominator is repeatedly subtracted from the numerator until the remainder is less than the denominator.
- "euclid" implements Euclid's greatest common divisor algorithm. Two numbers are read and saved. At each cycle the smaller number is subtracted from the larger number until they become equal.

Example	State Variables		Input Variables		Internal Variables		Sequential
	Boolean	Integer	Boolean	Integer	Boolean	Integer	Depth
ticket	6	4	1	0	43	6	∞
perfect	0	4	1	1	3	13	∞
sdiv	0	4	1	2	1	8	∞
euclid	0	4	1	2	2	8	∞
bound	6	6	1	6	25	10	5
movavg$n$	n	$n+1$	0	1	1	$2n+2$	$2n$
shiftbool$n$	n	0	0	0	0	0	n
shiftint$n$	0	n	0	0	0	0	n
shifteq$n$	0	n	0	0	n	0	n

Table 1. Characteristics of circuits.

- "bound" is the EFSM shown in Figure 1. It reads the x, y coordinates of two points and a difference vector, and checks whether the two points are closer than that difference. The control states are one-hot encoded.
- "movavg$n$" reads in a stream of numbers and keeps the sum of the last n numbers read. It has n registers to store the stream of inputs, and uses a one-hot control word to keep track of which register to update next.
- "shiftint$n$" is an integer circular shift register of length n. It has no inputs, and its initial state is $1, 0, \ldots, 0$.
- "shiftbool$n$" is exactly like shiftint$n$, except that its variables are Boolean, rather than integer. Its initial state is TRUE, FALSE, ..., FALSE.
- "shifteq$n$" is an integer circular shift register where, for a given register, if it holds a 0, then a 0 is passed, else a 1 is passed. Thus, for any initial state, after one step, all registers will contain either 0 or 1, and the behavior thereafter is like a pure circular shift register. The initial state is $1, 0, \ldots, 0$.

We had to address the treatment of negative integers, since Shasta and Omega differ on this point. Rather than encumbering Shasta by extending it to negatives, or burdening Omega by adding "$\geq$ 0" constraints on each variable, we decided to let each run in its "natural" mode. All of the examples were originally conceived as operating on the naturals; when presented with negative input values, some of the examples (e.g., sdiv) do not compute meaningful results, but we feel that Omega does not have to work "harder" because of this.

As mentioned in Section 3, Omega does not support Boolean variables as a special type. We experimented with two different encodings for Boolean variables for Omega: 1) FALSE is 0 and TRUE is $\neq$ 0, and 2) FALSE is $\leq$ 0 and TRUE is $>$ 0. We found that the second gives better results. Also, for the Omega experiments, formula minimization was applied after every Presburger operation, except for building atomic formulas, by calling the Omega function "simplify" with arguments (2, 2).

4.3 Results and Discussion

Computation of reachable states for any particular EFSM design proceeds in two phases. First we build, starting from the netlist representation of the EFSM, a single Presburger formula that defines its transition relation. Columns 3–6 in Table 2 show the CPU time (in seconds) and memory costs (in kilobytes) for this phase of the computation for each example, for both Shasta and Omega. The second phase of the computation is the iterative accumulation of reachable states, starting with an initial state, computing images, and checking for a fixed point. In those designs where a fixed point exists we let the computation proceed to that fixed point. In those designs where a fixed point does not exist, we run the computation out to where computational costs have grown significantly. Columns 7–10 in Table 2 show the costs for this phase of the computation.

Example	Depth	Build Transition Relation				Reachability			
		Shasta		Omega		Shasta		Omega	
		Time	Mem.	Time	Mem.	Time	Mem.	Time	Mem.
ticket	10	17.5	721	1061.5	16335	10.6	0	308.2	0
perfect	20	6.8	582	2.3	1507	36.3	1483	2636.9	11043
sdiv	40	1.8	25	0.4	672	2953.8	65872	414.6	3891
euclid	6	2.3	90	0.5	762	275.8	60834	83.4	2908
bound	5	196.0	46596	34961.6	29516	241.4	0	253.8	8602
movavg2	4	1.9	197	0.9	745	0.7	0	0.4	0
movavg3	6	3.6	950	3.9	1221	3.2	0	1.8	0
movavg4	8	6.7	2417	16.7	1909	13.3	16	6.0	0
movavg5	10	13.9	5956	74.5	3408	49.5	25	18.6	0
movavg6	12	43.5	16253	470.4	7160	208.8	0	51.3	0
shiftbool6	6	0.5	66	4.0	1376	0.8	8	1.7	434
shiftbool7	7	0.6	74	17.6	2630	1.2	8	3.5	1360
shiftbool8	8	0.8	74	80.1	5186	1.7	25	8.0	2777
shiftbool9	9	0.9	90	377.7	10420	2.3	25	17.6	5603
shiftbool10	10	1.0	98	1751.2	21266	3.0	41	41.6	11248
shiftint12	12	0.9	123	0.2	541	4.8	57	146.5	2048
shiftint13	13	1.1	147	0.2	590	6.0	66	238.2	2531
shiftint14	14	1.2	147	0.2	623	7.2	74	372.3	3097
shiftint15	15	1.3	180	0.2	655	8.8	74	550.7	3711
shiftint16	16	1.4	205	0.2	696	10.7	74	785.5	4391
shifteq4	4	1.2	295	6.1	1204	0.8	41	0.9	303
shifteq5	5	2.5	786	66.5	3039	4.8	106	3.0	1622
shifteq6	6	8.7	3195	742.4	9126	33.7	254	11.6	5292

Table 2. Results on building transition relations and performing reachability.

All experiments were run on a Sun Ultrasparc 3000 with a 168 MHz clock and 512 MB of main memory. The CPU times shown are as reported by the standard UNIX function "time". Transition relation build times do not include the time to read the input files. The memory costs shown are not very accurate, especially for reachability. We use the UNIX "sbrk(0)" function to determine the highest address of allocated memory before and after a function, and report the difference. This fails to account for the use of recycled free memory (this explains the 0KB figures in the table), and may also include memory speculatively allocated but not actually used.

Neither Shasta nor Omega emerge as consistently superior. In building transition relations, we never observed Omega running significantly[3] faster than Shasta. On the other hand, there are examples where Shasta runs much faster than Omega (ticket, bound, movavg$n$, shiftbool$n$, and shifteq$n$).

For the reachability computation itself the results are more mixed. There are examples where Shasta runs much faster than Omega; for other examples, Omega runs much faster than Shasta. In particular for the euclid design, Shasta could complete only 6 steps (due to memory use), while Omega could complete eight steps (we show the costs through step six for both tools). Considering both the model build and reachability phases together, Shasta is significantly faster on shiftbool$n$ and ticket, while there are no examples where Omega is significantly faster for both phases.

The above analysis compares the absolute runtimes of the two engines. By varying the reachability depth on the fixed-sized examples, and by varying the circuit size on the parameterized examples, we are able to empirically estimate the asymptotic performance of the two tools (Table 3). Overall, Shasta has the same or better (perfect, shiftbool$n$ and shiftint$n$) asymptotic performance in all cases analyzed.

Example	Phase	Shasta	Omega	Function of	see Figure
euclid	reach.	exponential	exponential	reachability depth	4a
sdiv	reach.	cubic	cubic		4b
perfect	reach.	quadratic	quintic		4c
movavg$n$	build	exponential	exponential	circuit size	4d
	reach.	exponential	exponential		
shiftbool$n$	build	exponential	exponential		
	reach.	exponential	exponential		
shiftint$n$	build	linear	linear		
	reach.	quadratic	exponential		
shifteq$n$	build	linear	exponential		
	reach.	quadratic	exponential		

Table 3. Asymptotic performance.

One conceivable factor that could give Shasta an advantage is its special treatment of Boolean variables. If we focus on the build phase, the presence of a significant number of Boolean state or internal variables (examples ticket, bound, movavg$n$, shiftbool$n$, shifteq$n$) is a perfect predictor of when Shasta outperforms Omega. However, for reachability, the presence of Boolean state variables does not assure Shasta is better: Shasta is faster for ticket and shiftbool$n$, but is the same or slower for bound and movavg$n$. Furthermore, Shasta's superiority in reachability on shiftbool$n$ cannot be explained simply by the presence of Boolean variables, since it similarly outperforms Omega on shiftint$n$, which has no Boolean variables. Recently, Bultan *et al.* [7] proposed a variant of Omega, where a Presburger formula with integer and Boolean variables is represented by a set of Omega and BDD pairs. They observed a drastic improvement over

[3] For runtimes of more than 7 seconds, "significantly" means, here and throughout, more than a factor of 3.

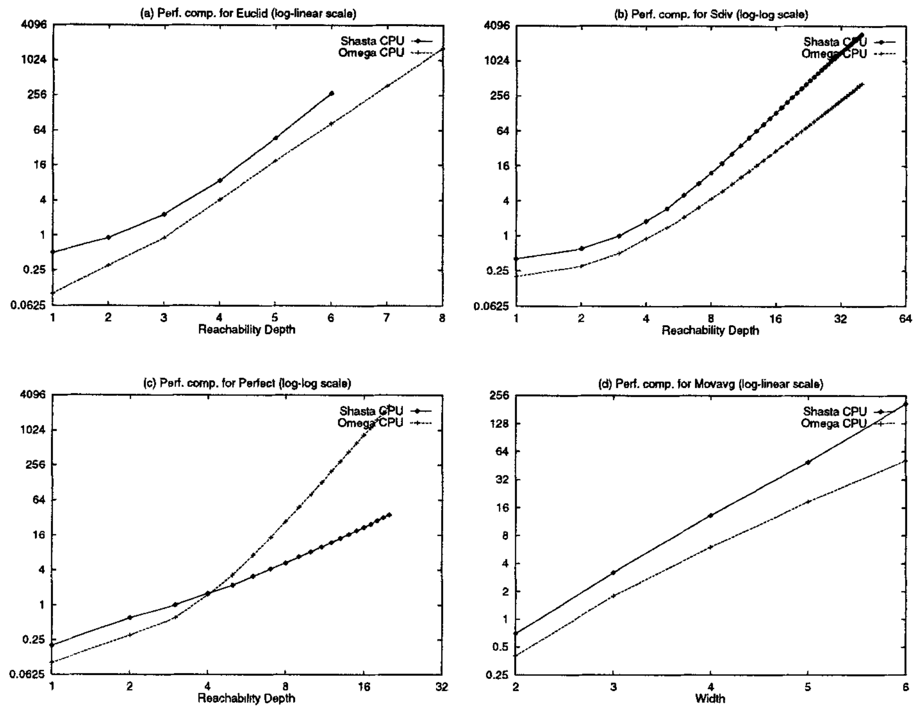

Fig. 4. Various plots comparing the performance of Omega vs. Shasta. The vertical axis is CPU time in seconds. Note that graphs a and d are log-linear, and b and c are log-log.

the standard Omega tool on the one example they studied; it would be instructive to perform a direct comparison within our framework.

Does the implementation of Shasta reflect the true potential of the automata-based approach? Fortunately, we were able to answer this question to some degree by comparing Shasta directly to Mona, a second-generation automata package that supports the logic WS1S [2, 14]. Rather than trying to integrate Mona into VIS, we just manually coded two examples (sdiv and euclid) in Mona's WS1S language, using the embedding suggested by Büchi. We also hardcoded the reachability computation out to a fixed number of steps for each example. By ensuring the same variable ordering, we were able to exactly match the automata (with BDD transitions) built by both Shasta and Mona. A performance comparison between Shasta and Mona on these two examples revealed that Mona consistently outperforms Shasta by almost a factor of 2 in runtime. In conclusion, even though the runtimes of Shasta could be reasonably halved, this does not fundamentally alter the observations made above in comparing Shasta to Omega.

5 Conclusions

Our research is focused on performing implicit state enumeration of EFSMs. The heart of such an approach is a computational engine for deciding the validity of Presburger formulas. Having found two very different types of engines discussed in the literature, we performed a set of experiments to discover which type of engine would work better for our application. Despite the different approaches taken by the automata-based Shasta and polyhedra-based Omega packages, their overall performance on our experimental EFSM reachability problems was remarkably similar. In absolute terms, these packages were able to analyze small designs with up to roughly a dozen state variables; the complexity of the Presburger decision problem prevents their application to much larger problems.

While neither package is consistently superior, we do observe large differences in performance, with each tool sometimes faster than the other by factors of more than 50. If it were possible to predict which approach would work better on which problem, it might be possible to build a hybrid engine that would outperform either package. We found that Shasta consistently builds the transition relation faster when Boolean variables are present, but this advantage does not always carry over to reachability. In the absence of a reliable predictor for performance, a simple hybrid approach, where both tools are applied and the slower tool aborted when the faster tool has finished, could be practical, since the performance differences between the tools can be so large.

There is a rough analogy between these two Presburger packages and approaches used in the Boolean domain. The automata-based Shasta package resembles a BDD package, while the polyhedra-based Omega package resembles a SOP Boolean function package. In the Boolean domain, BDD packages are widely accepted as the superior technology for general Boolean function representation. However in the Presburger domain, the BDD-like Shasta package does not enjoy such a clear cut practical advantage. In addition, while we have not observed any cases where Shasta has worse asymptotic performance, in the Boolean domain cases are known to exist where a SOP representation is superior to BDDs [11]. Thus, we expect that similar situations exist in the Presburger domain where Omega will outperform Shasta not only in raw terms, but also in asymptotic performance.

Acknowledgments We thank Kurt Keutzer for suggesting to us the use of Presburger arithmetic for analyzing EFSMs, and Adnan Aziz for modifying VIS to support EFSMs.

References

1. T. Amon, G. Borriello, T. Hu, and J. Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Proc. 34th Design Automat. Conf.*, pages 226–237, June 1997.
2. M. Biehl, N. Klarlund, and T. Rauhe. Mona: Decidable arithmetic in practice. In B. Jonsson and J. Parrow, editors, *Fourth International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, Uppsala, Sweden, 1996. Springer-Verlag.
3. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In D. L. Dill, editor, *Proc. Computer Aided Verification*, volume 818 of *LNCS*, pages 55–67, Stanford, CA, June 1994. Springer-Verlag.
4. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees and Algebra in Programming - CAAP*, volume 1059 of *LNCS*, pages 30–43. Springer-Verlag, 1996.
5. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In R. Alur and

T. A. Henzinger, editors, *Proceedings of the Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 428–432, New Brunswick, NJ, July 1996. Springer-Verlag.

6. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congress Logic, Methodology, and Philosophy of Science*, pages 1–11, Berkeley, CA, 1960. Stanford University Press.
7. T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 International Symposium on Software Testing and Analysis (ISSTA '98)*, 1998.
8. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state programs using Presburger arithmetic. In O. Grumberg, editor, *Proc. Computer Aided Verification*, volume 1254 of *LNCS*, pages 400–411, Haifa, June 1997. Springer-Verlag.
9. K.-T. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th Design Automat. Conf.*, pages 86–91, June 1993.
10. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373. Springer-Verlag, June 1989.
11. S. Devadas. Comparing two-level and ordered binary decision diagram representations of logic functions. *IEEE Trans. Computer-Aided Design*, 12(5):722–723, May 1993.
12. S. Devadas, K. Keutzer, and A. Krishnakumar. Design verification and reachability analysis using algebraic manipulation. In *Proc. Int'l Conf. on Computer Design*, pages 250–258, Oct. 1991.
13. H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.
14. J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95*, volume 1019 of *LNCS*, pages 89–110. Springer-Verlag, May 1995.
15. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library (Version 1.1.0) interface guide. http://www.cs.umd.edu/ projects/omega, Nov. 1996.
16. D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, July 1978.
17. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.
18. B. L. van der Waerden. *Modern Algebra*, volume 1. Ungar, 1953.
19. P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. of Static Analysis Symposium*, volume 983 of *LNCS*, pages 21–32. Springer-Verlag, Sept. 1995.