Implicit State Enumeration for FSMs with Datapaths

James H. Kukula
Synopsys, Inc., Beaverton, OR
kukula@synopsys.com

Thomas R. Shiple   Adnan Aziz*
Synopsys, Inc., Mountain View, CA
{shiple,adnan}@synopsys.com

October 10, 1997

DAC '98 Submission

Abstract

We show how the classic BDD-based technique of implicit state enumeration for FSMs can be generalized to an automata-based approach for implicit state enumeration of FSMs interacting with datapaths of unbounded width. We present experimental results showing that our automata representation of an unbounded width datapath can be 10x more compact than the BDD representation of the corresponding 32-bit datapath.

1 Introduction

The BDD-based approach to functional verification of finite state systems has enjoyed some success. This approach proceeds by first building BDDs for the next state functions and then using implicit state enumeration to explore the state transition graph [12]. For systems with up to a few hundred flip-flops this technique can be vastly more efficient than techniques that manipulate states explicitly. However, this approach usually fails for systems with wide numeric datapaths because the BDDs become too large.

We focus on systems whose datapaths consist of addition and comparison operations on integer data. The control component of such systems determines what operations to perform on the data, and in turn the results of data comparisons influence the control component. Our aim is to discover whether the system can reach any undesirable states; we do this by performing state reachability analysis. A control component operating alone in an unconstrained environment can in general reach more states than when constrained by interaction with a datapath. Thus it is important to analyze the system as a whole, to avoid erroneously reporting states being reachable when in fact they are not.

Our goal is to verify real digital systems, whose datapaths have fixed finite widths. To verify such systems, we generalize them to systems having datapaths with unbounded width. At first glance this generalization from finite width datapaths to unbounded width datapaths would only seem to make the verification problem more difficult. However, as we show, in some cases verifying the more general case can actually be more efficient. The more general case may also better model the user's actual verification problem. For example, a designer might construct a parameterized

*Visiting from the University of Texas, Austin

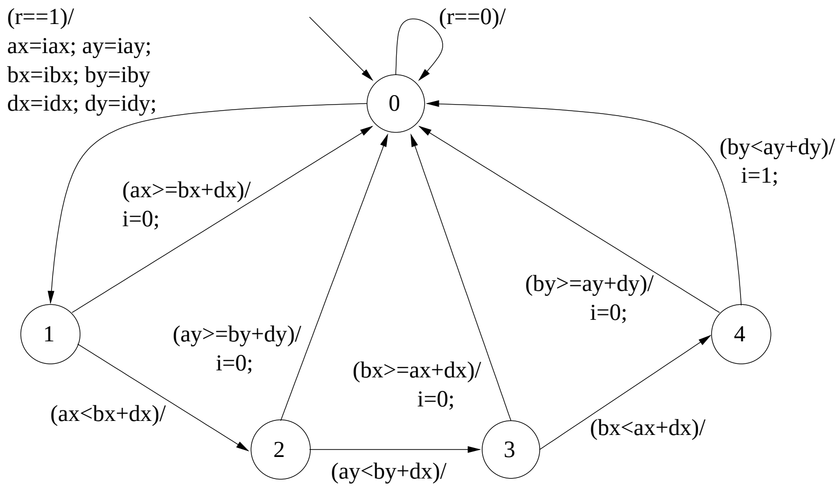

Figure 1: An EFSM.

library component that can be instantiated with arbitrary width. One would like to verify all possible instantiations.

Specifically, we model digital systems as extended finite state machines (EFSMs) [11], which are FSMs interacting with datapaths. In EFSMs, the state variables are partitioned into control variables (variables over finite domains) and data variables (unbounded integers). The behavior of an EFSM is defined by a set of transitions, each of which updates the values of both control and data variables. A transition has individual control states as origin and destination, but can specify the machine's behavior for whole sets of data states. Each transition has a gating predicate over the data variables, which must be satisfied for the transition to fire, and an update function that defines how the data variables change when the transition fires. The EFSM model naturally captures a frequently used machine structure, where a small, irregularly structured control component interacts with a large but regular datapath component. Note that since data variables are unbounded integers, the state space of an EFSM is infinite.

Figure 1 depicts a simple EFSM with five control states (represented graphically by circles), seven data variables ($a_x, a_y, b_x, b_y, d_x, d_y, i$), and ten transitions (represented graphically by arrows between circles). This machine reads data from its environment and then checks a series of inequalities that determine whether the variable i should get a 0 or 1 value.

BDDs are commonly used to represent sets of states of finite state machines. We need a different mechanism to handle the possibly infinite sets of states of EFSMs. The theory of formal languages deals with infinite sets, in particular with infinite sets of strings. With this in mind, we map states of EFSMs to strings, and view sets of states as languages. Furthermore, it is well known that regular languages can be represented by finite state automata; since the sets of EFSM states we consider form regular languages, we can represent them by automata. Also, since a relation over states can be viewed as a set of pairs of states, the transition relation of an EFSM can also be viewed as a set of strings, and represented by an automaton.

Note that, while EFSMs and language-recognizing automata are both state transition systems, we use them in very different ways. EFSMs are the machines we wish to verify, for example a signal processing chip that reads in raw data and writes out filtered data. In order to verify the signal processing chip, we generalize its datapath to unbounded width. We represent a single state of the chip by a string, and a set of states by a set of strings, *i.e.,* by a language. To compactly represent this language we use the automaton that recognizes it. This automaton reads in strings that represent states of the signal processing chip. If the automaton accepts a string, then the state is in the set we are considering, otherwise not. So in summary, EFSMs are the systems we are verifying, and automata are just a data structure used to represent sets of EFSM states and transitions.

We use implicit state enumeration to check whether an EFSM can reach any undesirable states. For finite state systems, implicit state enumeration can be performed by means of various BDD operations such as intersection and existential quantification. In a completely parallel manner, we perform implicit state enumeration for EFSMs by means of similar operations on automata. In fact, this parallel is strong since BDDs can be thought of simply as the special case of acyclic automata.

There is of course one difficulty in moving from finite systems to infinite systems. In general, state reachability for infinite systems is undecidable, so the reachability calculation may not converge in any finite number of steps. Nonetheless, we show examples where in fact fixed points are reached. And even when a fixed point cannot be reached, all bugs discovered will be true bugs, and any bug that exists can be discovered in finite time. The difficulty is that the absence of a bug cannot always be decided in finite time.

The main contribution of this paper is the generalization of the BDD-based approach for verification of finite state systems, to an automata-based approach for verification of EFSMs. Moving to EFSMs has two potential advantages. First, it is often more efficient to verify the EFSM; our results show reductions of 10x and more over BDDs for 32-bit datapaths. Second, one may want to verify a design for all possible widths of a datapath.

The same automata-based approach can be applied to iterated systems, which are systems with arrays of identical FSMs. Although this application of automata was noted recently by other researchers [17], we present here the first experimental evidence of the efficacy of this approach.

In Section 2 we discuss related work on the analysis of EFSMs. In Section 3 we introduce the EFSM model, discuss the use of automata to represent the states and transitions of EFSMs, and provide experimental results comparing the BDD approach to the automata approach for state reachability. In Section 4, we discuss the application of the automata approach to iterated systems. We conclude with a summary, evaluation, and future plans in Section 5.

2 Related Work

There has been much activity over the last decade in the analysis of EFSMs and iterated systems. Cheng and Krishnakumar formally defined EFSMs with data variables that constitute an n-dimensional linear space [11, 18]. Their first paper addresses functional test generation while the second discusses the computation of reachable states using finite unions of convex polyhedra as the underlying representation.

One approach taken in the analysis of EFSMs, and in the related area of hybrid systems, is to represent sets of data values implicitly (typically with polyhedra or automata), but to represent control states explicitly [6, 1, 5, 4]. In all of these works, state enumeration proceeds by visiting the control states explicitly, one-by-one, accumulating implicitly the data values possible at each control state. This is in contrast to our method where the control and data values are represented and manipulated uniformly.

There have been several attempts to integrate the representation of control and data values, but they all differ from our work in at least one respect. Asarin *et al.* use BDDs to represent the finite number of values that clocks can assume in discrete timed automata [2]; however, the sizes of the BDDs are sensitive to the number of bits needed to represent the clock values. In contrast, our analysis is insensitive to the bit widths of the data variables, since these variables are generalized to unbounded integers. Godefroid and Long use an extension of BDDs, called QBDDs, to analyze systems containing unbounded queues [13]. Their integrated handling of control and data in the context of queues motivated us to search for a similar mechanism in the context of EFSMs. Finally, by treating control variables as special cases of integers, Bultan *et al.* are able to treat control and data uniformly, using polyhedra as the underlying representation [10]. In practice, to avoid an explosion in the polyhedral representation, they distinguish control and data and explicitly visit control states one-by-one.

One of the key components in our approach is a method due to Boudet and Comon to translate an arbitrary system of linear relations into an automaton recognizing integer tuple solutions of the system [7]. However, they do not apply their work to the analysis of EFSMs. We perform implicit state enumeration on any EFSM whose transition relation can be described in Presburger arithmetic.

There has been a large body of work on verification of iterated systems, but since this is not the primary focus of this paper, we do not attempt to characterize all of this work. We were strongly influenced by the work embodied in the MONA tool, which translates formulas in the monadic second-order logic over strings into automata [3]. The authors show how this machinery can be used in a deductive framework to perform equivalence checking on iterated systems. The tool MOSEL follows the same approach as MONA [16]. Closely related is the work by Gupta [14]; she introduces linear inductive functions, which are sequential counterparts to BDDs, to represent iterated systems. She uses LIFs to perform implicit state enumeration on these systems by successively unrolling the systems in time.

The framework we use for the verification of iterated systems is the same as that independently reported by Kesten *et al.* [17]. Both our work and theirs build on the automata representation for iterated systems of Henriksen *et al.* [15]. However, whereas Henriksen uses a deductive framework, we and Kesten perform implicit state enumeration, generating each successive set of reachable states directly from prior sets of reachable states, without unrolling the system or introducing growing numbers of variables. We provide the first experimental results of this approach.

3 Implicit State Enumeration for EFSMs

3.1 EFSM Model

As mentioned in the introduction, conceptually the variables of an EFSM are partitioned into control and data. However, we can treat finite domain control variables as special cases of infinite domain data variables. This allows us to treat all variables uniformly, simplifying the formalism.

Thus, we formally define an EFSM to be a tuple (k_i, k_o, k_s, I, T), where[1]

- $k_i \in N$ gives the number of input variables,
- $k_o \in N$ gives the number of output variables,

[1] N represents the set of natural numbers, and B the set $\{0, 1\}$.

- $k_s \in N$ gives the number of state variables,
- $I : N^{k_s} \to B$ is the characteristic function of the set of valid initial states, and
- $T : N^{k_i} \times N^{k_s} \times N^{k_s} \times N^{k_o} \to B$ is the characteristic function of the transition relation for the machine. $T(x_i, x_s, x'_s, x_o)$ is true iff the EFSM, upon receiving input x_i, can make a transition from state x_s to state x'_s producing output x_o.

The transition relation T encodes both the transition gating predicates over the data variables and the update functions for the data variables. For example, consider the transition from control state 1 to 0 in the EFSM shown in Figure 1, and let s be the control state variable. This transition is represented by the conjunction of the present control state value, the gating predicate, the next control state value, and the update function, as follows:[2]

$$
\begin{aligned}
&(s = 1) \wedge (a_x \geq b_x + d_x) \wedge (s' = 0)\\
&\wedge(i' = 0 \wedge a'_x = a_x \wedge a'_y = a_y \wedge b'_x = b_x \wedge b'_y = b_y \wedge d'_x = d_x \wedge d'_y = d_y)
\end{aligned}
$$

The entire transition relation T is just the disjunction of the formulas for each of the 10 transitions of this EFSM.

We address the problem of computing the set of reachable states of EFSMs (over both the control *and* data variables). Our goal is to perform state enumeration implicitly. To show how this can be done, we must first introduce the relationship between EFSMs and Presburger arithmetic, and the relationship between Presburger arithmetic and automata.

3.2 Representing EFSM States and Transitions

In order to compute the reachable states of an EFSM, we need an effective way to represent and manipulate sets of states and relations between states. An overly rich set of arithmetic operators for the gating predicates and update functions would make calculations too difficult, while an overly constrained set would make it impossible to treat many practical EFSMs. As a compromise, we restrict our attention to EFSMs whose gating predicates and update functions are definable in Presburger arithmetic. Presburger arithmetic is the first-order theory of natural numbers with comparison and addition,[3] and specifically without multiplication. For every Presburger formula there is a finite automaton that recognizes the integer tuples that satisfy the formula [9].[4] This link between finite automata and Presburger arithmetic is what makes automata useful for verification of EFSMs. By using automata to represent Presburger formulas, we can perform implicit state enumeration for EFSMs whose gating predicates are Boolean combinations of linear equations and inequalities, and whose update functions are linear (affine) mappings.

To see how automata can represent Presburger formulas, first consider regular languages consisting of strings over the alphabet of bit vectors B^k. These are the languages recognized by finite automata that read bit vectors. An example of such a machine, for $k = 2$, is shown in Figure 2. This machine recognizes the language

$$
\left(\begin{matrix} 0 \\ 1 \end{matrix} + \begin{matrix} 1 \\ 1 \end{matrix} \right) \begin{matrix} 1 \\ 0 \end{matrix} \left(\begin{matrix} 0 \\ 0 \end{matrix} \right)^*
$$

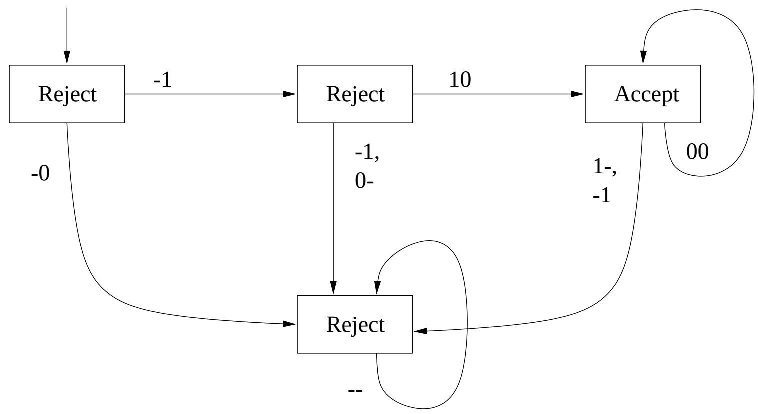

Figure 2: An automaton. "-" means "0 or 1".

Given automata representing languages f and g, there are well known algorithms to build new automata to represent the sets $\overline{f}$, $f \vee g$, and $f \wedge g$ [20]. We can also existentially quantify particular components of the bit vectors, forming an automaton with an alphabet of smaller dimension. For example, existentially quantifying the first component of the above language yields a new language in only one dimension, the language 100^*. As we will see, these operations are the basic building blocks of implicit state enumeration. Also crucial is the fact that the minimum state automaton for a language is unique, and hence is a canonical form.

Each transition of an automaton must specify for which inputs the automaton is to follow that transition. When the letters of the alphabet are bit vectors with large dimension, the transition labels can get very complex. Following [15], we use BDDs with multiple terminals as a practical implementation technique to efficiently represent the transitions of an automaton. 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 in Figure 2, with its transitions represented by BDDs.

Now that we have an efficient method to represent automata that read bit vectors, we can discuss how to use automata to represent the set of integer tuples satisfying a Presburger formula. Given a bit string $b = b_0 b_1 \ldots b_{n-1} \in B^*$, we can interpret it as a natural number with low order bits first:

$$x(b) = \sum_{i=0}^{n-1} b_i 2^i$$

This mapping is easily extended to multiple dimensions, $x : (B^k)^* \rightarrow N^k$. For example, the string of bit vectors

$$\begin{array}{lccccc}
 & \multicolumn{4}{c}{\text{string} \longrightarrow} & \\
x_1: & 0 & 0 & 1 & 0 & \text{bit} \\
x_2: & 1 & 1 & 1 & 0 & \text{vector} \\
x_3: & 1 & 1 & 0 & 1 & \downarrow
\end{array}$$

[2]The input is a don't care on this transition. The data variables $a_x, a_y, b_x, b_y, d_x, d_y$ keep their previous values.

[3]In fact, other operators, such as (mod n) can be defined in Presburger arithmetic, in terms of the primitives.

[4]We use the term "integers" loosely to refer to natural numbers.

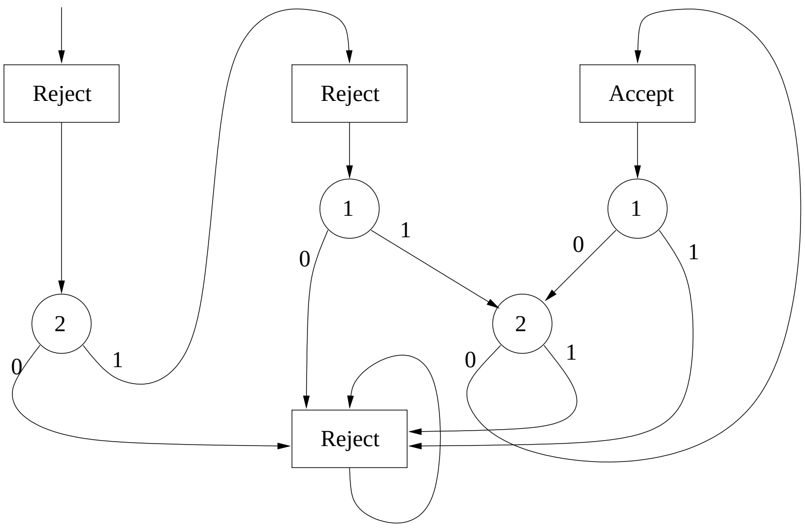

Figure 3: Using BDDs for transitions of automata.

represents the integer tuple $(x_1, x_2, x_3) = (4, 7, 11)$.

Using this encoding for natural numbers, Boudet and Comon have given a simple procedure for constructing automata from Presburger formulas [7]. The constructed automaton accepts just those integer tuples that satisfy the corresponding formula. Figure 4 shows the automaton that recognizes the integer tuples satisfying the Presburger formula $x_1 + x_2 = x_3$. To illustrate its operation, consider again the integer tuple $(4, 7, 11)$, represented by the string of bit vectors shown above. When reading this string, the automaton starts at the initial state. The first input has 0 for x_1, 1 for x_2, and 1 for x_3, which leads the automaton back to the initial state, ready to read the second bit vector in the string. After reading all the bit vectors, the automaton will end in the accepting state, reflecting the fact that $4 + 7 = 11$.

3.3 Implicit State Enumeration

Given an EFSM, we want to calculate the set of reachable states $R \subseteq N^{k_s}$. We use the usual iterative approach:

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

The sequence of R_j is monotonically increasing but may not converge, since we are dealing with infinite state systems. However, all states x_s for which any $R_j(x_s)$ is true are indeed reachable, so if any of these were not intended to be reachable, then a true design error has been exposed.

The foundation of our approach is the observation that, if I and T are Presburger formulas, then the various R_j are also Presburger formulas. When the datapath of an EFSM is composed of addition and comparison operations acting on natural numbers, I and T can indeed be expressed as Presburger formulas. We use automata to represent Presburger formulas, so the iterative calculation of reachable states is performed by union, intersection, and existential quantification of automata.

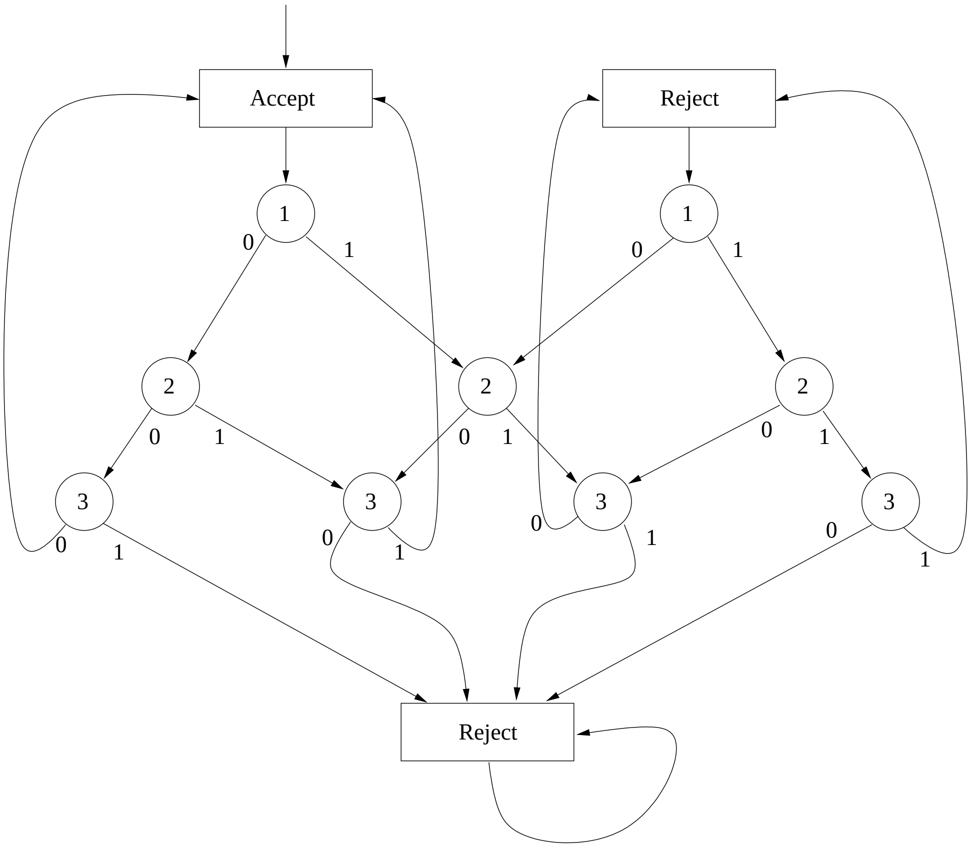

Figure 4: The automaton representing $x_1 + x_2 = x_3$.

An efficient minimization procedure generates canonical forms of the resulting automata. This keeps the representation as small as possible and allows testing for convergence to a fixed point.

The real systems we wish to verify have finite fixed width datapaths. The states of these systems can simply be mapped to strings of a finite fixed length. Once we have an automaton R representing the set of reachable states for the unbounded EFSM, to decide if a given real state is reachable we can just check if R accepts the string that represents the real state.

In order for this computation to be accurate, the transition relation T must accurately model the real system. The most common difficulty in building an accurate model is arithmentic overflow. Addition in an 8-bit system could be modeled in an EFSM by the formula

$$(x + y < 256 \wedge z = x + y) \vee (x + y \geq 256 \wedge z = x + y - 256)$$

The effect of this, however, is to limit the computation on the EFSM to systems of a single width, which in fact unrolls the automata and results in computations isormorphic to BDDs.

In systems where overflow can occur, one can instead introduce a new parameter variable to specify the bound:

$$(x + y < b \wedge z = x + y) \vee (x + y \geq b \wedge z = x + y - b)$$

With this form of the transition relation, one is again representing an infinite class of systems and the compact looping structure of the automata is preserved. When checking for the reachability of a state of a real fixed width system, one represents the state by a string that includes the actual value of the bound. In the experiments that follow, none of the circuits exhibit rollover, so the modelling as EFSMs was straightforward.

3.4 Experimental Results

The purpose of our experiments is to compare the BDD-based approach to state reachability of systems with fixed width datapaths to the automata-based approach to state reachability for the corresponding systems with unbounded width datapaths. We developed a package, written in C, to manipulate automata with BDD transitions. We incorporated this into the VIS program [8], along with a front-end package that interprets a dialect of Verilog that includes wires carrying unbounded integer values and arithmetic modules that operate on them. We used the standard VIS program to characterize the BDD approach to state reachability for fixed width datapaths. We fixed the variable ordering in VIS to correspond to that used in the automata approach, by interleaving the datapath variables and putting the low order bits first; the control variables always precede the data variables. All experiments were run on a Sun Ultrasparc 3000 with a 168 MHz clock and 512 MB of main memory.

We developed several small designs with datapaths; these are typical circuits found in DSP, communication protocol, and computer applications. The experimental circuits are briefly characterized in Table 1; *control variables* refers to the number of binary variables, *data variables* to the number of integer variables, and *sequential depth* to the greatest lower bound on the path length from the initial state to any reachable state.

- "movavg" reads in a stream of numbers and keeps the sum of the last eight numbers read. It keeps these numbers in data registers, and on each cycle overwrites with the current input the register with the oldest number. A one-hot coded control word keeps track of which data

example	control variables	data variables	sequential depth	
			fixed width w	unbounded width
movavg	9	9	15	15
sdiv	0	4	2^w	∞
bound	6	6	5	5
euclid	0	4	2^w	∞

Table 1: Characteristics of experimental circuits.

example	automata size	BDD size for bit width				ratio for 32 bits
		8	16	24	32	
movavg	560	3361	7249	11137	15025	26.8
sdiv	3407	2002	11911	22423	32935	9.7
bound	2380	7013	17045	27077	37109	15.6
euclid	19196	36408	172090	307914	443738	23.1

Table 2: Sizes of automata and BDDs for reached state sets.

register to update next. In each cycle the accumulator is decremented by the oldest number and incremented by the new number.

- "sdiv" is a serial divider. A numerator and denominator are read in, and then the denominator is repeatedly subtracted from the numerator. The sequential depth of this machine grows exponentially with the width of the datapath, and is unbounded for unbounded integers. The EFSM reachable state calculation thus does not reach a fixed point, nor does the BDD reachability calculation converge in a practical time even for moderate sized datapaths.

- "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 checks are performed one at time. The control states are one-hot encoded.

- "euclid" implements Euclid's greatest common divisor algorithm. Two numbers are read in and saved. These numbers are copied to two working registers. At each cycle the smaller number is subtracted from the larger number. This process halts when the two numbers are equal, at which point they will have the value of the GCD of the numbers initially read in. This machine has a sequential depth that grows exponentially with the width of the datapath.

Table 2 shows the BDD size and automaton size[5] for the reachable state set for a given step of reachability. For "movavg", data is given for step 8 (where the peak sizes are seen) and for "bound" step 5 (the fixed point and peak). For "sdiv", the sizes grow gradually—step 19 is shown as a representative example, and for "euclid" they grow quickly—step 6 is shown. For most of our

[5] For an automaton, the size is taken as the sum of the number of BDD nodes that represent the transitions, and the number of terminal nodes that represent the states of the automaton.

example	automata size	BDD size for bit width 8	16	24	32
movavg	68320	16642	34714	52786	70858
sdiv	180	1036	2252	3468	4684
bound	26096	3740	8236	12732	17228
euclid	290	1804	3980	6156	8332

Table 3: Sizes of automata and BDDs for monolithic transition relation.

example	automata cost	BDD cost for bit width 8	16	24	32
movavg	1533/ 59	15/ 11	47/ 19	91/ 33	150/ 36
sdiv	229/ 16	2/ 2	11/ 9	25/ 13	31/ 18
bound	379/ 49	6/ 8	17/ 14	29/ 22	43/ 32
euclid	252/ 63	9/ 8	46/ 22	153/ 36	174/ 51

Table 4: Computational costs for implicit state enumeration by automata and by BDDs for several datapath widths. (CPU sec. / Megabytes)

examples, the BDD representation of the reached state set for 8-bit datapaths is already larger than the automata representation, and continues to grow roughly linearly with the width of the datapath. At 32 bits the BDDs are roughly 10 times larger or more; the final column shows the ratio of BDD sizes for the 32-bit case to the automaton sizes.

The sizes of the transition relations for the different circuits are shown in Table 3. Here the automata-based representation does not show as consistent an advantage over BDDs as was shown for reachable state sets. Still, the BDD transition relations for 32-bit datapaths are larger in every case than the automata-based transition relation for unbounded width datapaths.

Notice that for a fixed w-width datapath, the BDD representing a state set or transition relation is never more than w larger than the corresponding automaton. This is not coincidental, since "unrolling" an automaton w times yields an acyclic automaton, essentially a BDD, representing the language of the original automaton restricted to length w strings. This fact strictly bounds the improvement that automata can achieve relative to BDDs.

Table 4 shows compute time and memory for building the transition relation and performing reachability. In all cases the automata approach takes significantly more resources. Much of this inefficiency is undoubtedly due to the unoptimized prototype nature of our code. But there are more fundamental underlying causes as well. While the looping structure of automata allows more compact representation, it can make manipulation less efficient. With conventional BDDs, the acyclic structure means that operations near the root can take advantage of lower level canonicalization. With automata, an entire operation must be completed before canonicalization can be performed. This can result in larger intermediate results during calculations. In addition, the cost of canonicalization for automata is $O(n \log n)$, somewhat worse than the linear cost of canonicalization for BDDs. Nonetheless, we believe that the reduction observed in representation size is indicative of

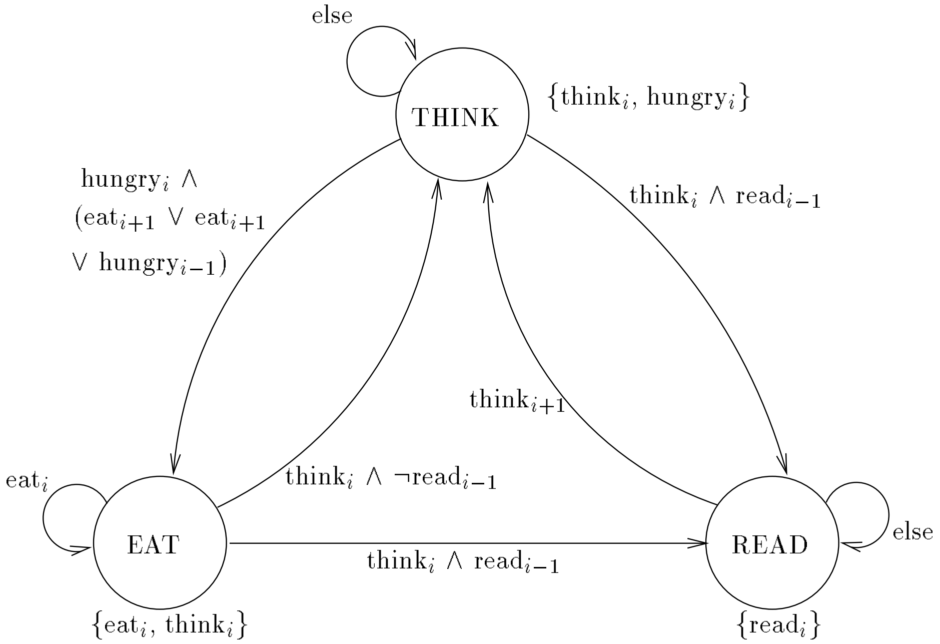

Figure 5: Transition relation for philosopher i. All arithmetic operations are modulo n.

the potential for runtime and memory improvements also.

4 Iterated Systems

In this section, we show how automata-based implicit state enumeration can be applied to iterated systems. An example of an iterated system, which we will use throughout this section, is the Dining Philosophers with Encyclopedia example of Kurshan and McMillan [19]. In this example, n philosophers sit in a ring; each philosopher can be in one of three possible states: THINK, EAT and READ. Philosopher i communicates with its left neighbor (philosopher $i-1 \bmod n$) and right neighbor (philosopher $i+1 \bmod n$). The transition relation for philosopher i is shown in Figure 5. After the n-philosopher system enters a new state, each philosopher nondeterministically selects an action, which is then used in determining the next state of the system. We would like to check the following two safety properties of the n-philosopher system, for all values of n:

1. exactly one philosopher is reading, and
2. no two neighboring philosophers are eating.

Whereas in EFSMs, we use strings to represent integers, here strings are used to represent states of arrays of philosophers. For example, for $n = 4$, the string EAT READ THINK THINK represents the state where the first philosopher is eating, the second philosopher is reading, etc.[6] Likewise, a triple of strings can represent a global transition, where the first string indicates the

[6]An encoding is used to reduce such strings to a set of bit strings.

present state, the second string the nondeterministic selection of philosopher actions, and the third string the next state. So for $n = 4$ the strings

EAT	READ	THINK	THINK
eat	read	think	hungry
EAT	THINK	READ	THINK

represent the global transition where the third philosopher selects "think" in state THINK and moves to READ, etc. Henriksen *et al.* show the formula that encodes the global transition relation, for all n, and they give a procedure to translate such formulas to automata [15]. Roughly speaking, the method used to create the global transition relation is to introduce a one-hot encoded variable i, define a single philosopher in terms of i, and then universally quantify i.

While Henriksen *et al.* use automata in a conservative, deductive style to prove that the above safety properties are inductive invariants, we apply reachability analysis on the global transition relation to show that the properties hold. Since our technique is exploring the states of an infinite number of finite state systems simultaneously, it is not guaranteed to converge in general. This is illustrated by the Dining Philosophers example. If the initial state is taken as philosopher 1 reading and all others thinking, then reachability never converges: since we are considering a system with an unbounded number of philosophers and because the encyclopedia can move at most one philosopher per system transition, the number of steps it takes to reach the farthest philosopher is unbounded.

On the other hand, if we consider an infinite set of initial states where any single philosopher is reading and all others are thinking, as Kurshan and McMillan suggest, then reachability converges in 3 steps! To emphasize, we perform exact implicit state enumeration on the global transition relation for an n-philosopher system, for all n; the computation converges in 3 steps yielding an automaton that recognizes all reachable states for an n-philosopher system, for all n. This automaton can then be tested for language inclusion against the automata representing the above safety properties. Thus, using a generalization of the classic implicit enumeration approach for finite state systems, we have automatically verified an infinite number of dining philosopher systems.

In Table 5, we compare the results of automata-based reachability, for the case of multiple initial states describe above, to the BDD-based reachability using VIS. The first column gives the number of philosophers for the BDD experiments, the second column the size of the monolithic transition relation, the third column the size of the final reachable state set, and the last column the time to build the transition relation and perform reachability. For the fixed systems, we order the variables of each philosopher together, and order the philosophers consecutively; we use the corresponding ordering for the parameterized system. As can be seen, it is more efficient to verify an n-philosopher system for all n than it is to verify a single n-philosopher system for fixed n greater than or equal to 12. Although the Dining Philosopher example itself is only of academic interest, it is suggestive of real-life systems such as a bus with many identical clients.

5 Conclusions and Future Work

We have shown how BDD-based implicit state enumeration of FSMs can be generalized to automata-based implicit state enumeration of infinite state EFSMs. Our experiments demonstrate that an automaton representation of an unbounded width datapath can be 10x more compact than the BDD representation of the corresponding 32-bit datapath. Using this same automata-based approach,

	# of phil.	size of trans. reln.	size of reach set	CPU time (seconds)
BDD-based	4	148	24	0.2
	8	606	120	0.5
	12	1066	232	1.0
	16	1526	344	1.2
	32	3366	792	3.5
	64	7046	1688	12.2
	128	14406	3480	47.5
	256	29126	7064	194.4
	512	58566	14232	963.1
aut.-based	all n	332	72	0.8

Table 5: Representation sizes and runtimes for reachability analysis on fixed Dining Philosopher systems, and the parameterized system.

we also presented experimental results for the Dining Philosophers iterated system that show that safety properties can be verified for an arbitrary n-philosopher system more efficiently than for a BDD-based approach for a fixed n greater than or equal to 12. These results point the way to a new approach for handling the state explosion problem caused by wide datapaths and large iterated systems.

Even though the automata approach achieves significant reductions in representation sizes, the run times tend to be much worse, for two reasons. The first is intrinsic: since automata have cycles, the graph algorithms have higher complexity. The second is that our implementation of the automata package is probably equivalent in sophistication to the first BDD packages of a decade ago. Just as significant improvements were achieved by more mature BDD packages, we believe that an order of magnitude performance improvement may be possible for our automata package. There are simplifications of data structures that should be straightforward. A very significant improvement would be provided by finding a way to perform partial state minimization of an automaton before the entire automaton is constructed. Developing such techniques is an important part of our future plans. Beyond improving the automata package, we would like to apply our techniques to more examples, and to add a temporal logic model checker.

References

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems I*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229, New Brunswick, NJ, 1993. Springer-Verlag.

[2] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In O. Maler, editor, *Proc. International Workshop on Hybrid*

and Real-Time Systems, HART '97, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360, Grenoble, Mar. 1997. Springer-Verlag.

[3] D. A. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In P. Wolper, editor, *Proc. Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 31–41. Springer-Verlag, July 1995.

[4] B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In O. Grumberg, editor, *Proc. Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 167–178, Haifa, June 1997. Springer-Verlag.

[5] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite states spaces using QDDs. In R. Alur and T. A. Henzinger, editors, *Proc. Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12, New Brunswick, NJ, July 1996. Springer-Verlag.

[6] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In D. L. Dill, editor, *Proc. Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67, Stanford, CA, June 1994. Springer-Verlag.

[7] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Trees and Algebra in Programming - CAAP*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.

[8] 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 *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, July 1996. Springer-Verlag.

[9] 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.

[10] 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 *Lecture Notes in Computer Science*, pages 400–411, Haifa, June 1997. Springer-Verlag.

[11] 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.

[12] 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 *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[13] P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. In *Proc. Logic in Computer Science*, pages 198–206, July 1996.

[14] A. Gupta and A. L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 192–199, Nov. 1993.

[15] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Larlund, 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 *Lecture Notes in Computer Science*, pages 89–110. Springer-Verlag, May 1995.

[16] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. MOSEL: A flexible toolset for monadic second-order logic. In *TACAS '97: Int'l Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 183–202. Springer-Verlag, Apr. 1997.

[17] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer-Verlag, June 1997.

[18] A. Krishnakumar and K.-T. Cheng. On the computation of the set of reachable states of hybrid models. In *Proc. 31st Design Automat. Conf.*, pages 615–621, June 1994.

[19] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proc. Eighth Symp. Princ. of Distributed Computing*, pages 239–247, 1989.

[20] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.