Verification and Synthesis of Counters based on Symbolic Techniques

Gianpiero Cabodi [†] Paolo Camurati [#] Luciano Lavagno [‡] Stefano Quer [†]

[†] Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

[#] Università di Udine
Dip. di Matematica e Informatica
Udine, ITALY

[‡] Politecnico di Torino
Dip. di Elettronica
Turin, ITALY

Abstract

Symbolic Techniques have undergone major improvements but extending their applicability to new fields is still a key issue. A great limitation on standard Symbolic Traversals is represented by Finite State Machines with a very high sequential depth. A typical example of this behaviour are counters. On the other hand systems containing counters, e.g. embedded systems, are of great practical importance in several fields. Iterative squaring can produce solutions with a logarithmic execution time with respect to the sequential depth but a few drawbacks usually limit its application. We successfully tailored iterative squaring to allow its application for symbolic verification and synthesis of circuits containing counters. Experiments on large and complex home–made and industrials circuits containing counters show the feasibility of the approach.

1 Introduction

Embedded systems are becoming more and more pervasive in every aspect of everyday life. They are currently implemented as a mix of hardware and software components. Hardware generally provides the performance required to meet real–time constraints. Software provides the flexibility required to re-use the system in a variety of situations. Recently, embedded system design [1] has become the target of a flurry of activity in the Computer Aided Design community. The aim of this activity, which is part of the more general research in hardware/software co–design, is to provide means to *validate* the specification against functional requirements and *synthesize* the hardware, the software and the interfaces.

Several timing verification problems can be reduced to reachability analysis of the composition of a system and one or more counters [2]. Moreover, modern micro–controllers provide the designer with several counter–based resources (like timers, serial I/O controllers and so on). Both synthesis and analysis of embedded software for such controllers then must rely on traversing efficiently extremely deep state spaces.

Previous work in this area has tackled mostly the problem of *abstracting* counters to non–deterministic Finite State Machine (FSMs) with fewer states [3], [4]. This approach can be applied only to a few cases, in which the actual number of states of the counter does not matter (e.g. proving safety or fairness constraints). Unfortunately, this is not the case for the applications listed above, in which the exact number is the key property.

On the other hand symbolic techniques reach their limits on FSMs with a very high sequential depth. In this case breadth first traversals reach few states (often a single one) at each iteration, thus requiring too many iterations (linear with the number of reachable states).

To solve this problem *iterative squaring* [5], [6], [7] plays an important role, because it can produce solutions with a *logarithmic* execution time with respect to the sequential depth of the FSM.

Unfortunately, a few drawbacks drastically limit the application of such a technique and our purpose is to extend its applicability tailoring iterative squaring to allow its application for the symbolic verification and synthesis of system containing counters. In the former case we use iterative squaring in a forward traversal approach, computing or verifying the duration of the counting phase (number of clock cycles before the terminal count). In the latter we resort to a backward approach; in this way we are able to determine the initial state to be loaded in the counter to reach a terminal count within a desired number of clock cycles. To deal with counters whose module is not a power of two, we have to use a dichotomic strategy.

Experimental evidence is reported on large and complex home–made and industrials circuits and shows the feasibility of the approach.

The remainder of the paper is organized as follows. Section 2 summarizes useful concepts. Section 3 describes the enhanced iterative squaring and Section 4 describes its application in verification and synthesis. Section 5 shows experimental results. Section 6 closes the paper with a brief summary and reports on future work.

2 Preliminaries

2.1 The Model

A *Finite State Machine* is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 6–tuple $M = (I, O, S, \delta, \lambda, S_0)$ where I is the input alphabet, O is the output alphabet, S is the state space, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function and $S_0 \subseteq S$ is the initial state set.

In the sequel we will use $s = (s_1, s_2, \ldots, s_n)$, $x = (x_1, x_2,$

ED&TC '97 on CD-ROM

$\ldots, x_m)$ and $y = (y_1, y_2, \ldots, y_n)$ to indicate respectively the vector of present state, primary input and next state variables.
BDDs will be used to represent functions, relations and sets.

2.2 The Transition Relation and Symbolic Traversals

The *transition relation* TR associated to a FSM M is:

$$\mathsf{TR}(s,x,y) = \prod_{i=1}^{n}(y_i \equiv \delta_i(s,x)) = \prod_{i=1}^{n} tr_i(s,x,y_i)$$

Very often the existence of an input, rather than its value, is important, then primary inputs may be abstracted from TR, defining a new relation T

$$\mathsf{T}(s,y) \quad = \quad \exists_x \mathsf{TR}(s,x,y)$$

usually called *non–deterministic transition relation.*
The image (pre–image) of a set of states described by its characteristic function $\mathsf{C}(s)$ ($\mathsf{C}(y)$) according to the transition relation is defined as:

$$\begin{array}{lcl} \mathbf{Img}(\delta(s,x),\mathsf{C}(s)) & = & \exists_{s,x}\ (\mathsf{TR}(s,x,y)\cdot\mathsf{C}(s)) \\ \mathbf{PreImg}(\delta(s,x),\mathsf{C}(y)) & = & \exists_{x,y}\ (\mathsf{TR}(s,x,y)\cdot\mathsf{C}(y)) \end{array}$$

The set of forward reachable state set R and the set of backward reachable state set BR can be computed with the two following fixed point computations:

$$\begin{array}{lcl} \mathsf{R}_i(y) & = & \mathsf{R}_{i-1}(y) + \mathbf{Img}(\delta, \mathsf{R}_{i-1})(y) \\ & = & \mathsf{R}_{i-1}(y) + \exists_{s,x}\ (\mathsf{TR}(s,x,y)\cdot\mathsf{R}_{i-1}(s)) \\ \mathsf{BR}_i(s) & = & \mathsf{BR}_{i-1}(s) + \mathbf{PreImg}(\delta, \mathsf{BR}_{i-1})(s) \\ & = & \mathsf{BR}_{i-1}(s) + \exists_{x,y}\ (\mathsf{TR}(s,x,y)\cdot\mathsf{BR}_{i-1}(y)) \end{array}$$

R_0 (BR_0) is set to the initial set of states. R_i (BR_i) is the reachable (backward reachable) state set at step i. In the sequel we will call R^* (BR^*) the values of B_i (BR_i) at the fixed point.
The number of iterations of the first recurrence equation gives the *sequential depth* of the machine, i.e. the maximum number of steps in the graph between the initial states and any other state.

2.3 Iterative Squaring

Although standard symbolic traversals are usually quite efficient they become impratical for very high sequential depth. In these cases a systematic method, called *iterative squaring* transformation [5], [6], [7], can result in an exponential reduction in the number of iterations necessary to reach fixed points.
$\mathsf{TR}(s,\ x,\ y)$ describes triples of the form $(\hat{s},\ \hat{x},\ \hat{y})$, where state $\hat{y}$ is reached from state $\hat{s}$ by applying input $\hat{x}$ whereas T describes pairs of adjacent states, i.e. states that are connected by at least one edge in the transition graph of M. The transitive closure T^* of T describes the pairs of states that are connected by at least one path in the state graph of M. T^* is found by computing the least fixed point of the following recurrence equations:

$$\mathsf{T}^{2^{i+1}}(s,y) \ = \ \mathsf{T}(s,y) + \exists_z\ (\mathsf{T}^{2^i}(s,z)\cdot\mathsf{T}^{2^i}(z,y)) \qquad (1)$$

setting $\mathsf{T}^1(s,y)$ to $\mathsf{T}(s,y)$[1]. The number of iterations required to compute T^* is logarithmic in the maximum distance between two states in the state transition graph (i.e. the diameter of the graph).
Finding the set of states reachable from S_0 either in forward or backward mode then amounts to computing:

$$\begin{array}{lcl} \mathsf{R}^*(y) & = & S_0(y) + \exists_s\ (T^*(s,y)\cdot S_0(s)) \\ \mathsf{BR}^*(s) & = & S_0(s) + \exists_y\ (T^*(s,y)\cdot S_0(y)) \end{array}$$

3 Improving Iterative Squaring

A few drawbacks drastically limit the application of iterative squaring:

- Computing TR and T is very often quite expensive or even impossible.
- The computation of T^* possibly blows up in few iterations due to the growing size of the BDDs.
- TR, T and T^* also consider paths originating at states that cannot be reached from the initial states. Hence squaring may be inefficient if a large fraction of the state graph is unreachable or if the sequential depth of the FSM is low.

For the above reasons iterative squaring is effective on "pure" counters, while this is not the case with other circuits. On the other hand, counters represent a difficult application field for standard traversal techniques, due to their high sequential depth. Taking into account these considerations, we adopt state–of–the–art squaring technique and modify it in the following ways.
When dealing with counters, we identify for them two behaviours or working phases: a programming and a counting phase. In the former phase data and control inputs are set to convenient values to force the counter to the desired initial state. Standard traversal techniques are well suited for this phase, usually requiring not more than a few breadth–first traversal steps. In the counting phase we need iterative squaring to limit the number of traversal iterations.
During the overall counting phase totally free sequences of counting steps can be avoided constraining some *control* inputs to be kept unchanged. Let us suppose to partition x variables in two sets: the constrained variables x_c and the free ones $x_f = x - x_c$; we compute:

$$\widehat{\mathsf{T}}(s,x_c,y) \quad = \quad \exists_{x_f}\ \mathsf{TR}(s,x,y)$$

The value of x_c can be fixed in each iteration of the traversal and this usually greatly simplifies the overall procedure. The following modified squaring formula describes the counting phase:

$$\widehat{\mathsf{T}}^{2^{i+1}}(s,x_c,y) = \widehat{\mathsf{T}}(s,x_c,y) + \exists_z(\widehat{\mathsf{T}}^{2^i}(s,x_c,z)\cdot\widehat{\mathsf{T}}^{2^i}(z,x_c,y))$$

We call this method *constrained computation.*
Moreover T^* can be computed incrementally; if c is a constraining function possibly related to knowledge of the circuits' behaviour we compute T^* as:

$$\mathsf{T}^* \quad = \quad (\mathsf{T} + (\mathsf{T}\cdot c)^*)^*$$

[1] The notation used in Equation (1) is not the standard one but we introduced it to allow the composition of terms not power of two.

This kind of computation, that we call *incremental squaring*, decreases complexity as it proceed incrementally. Generally the squaring can be computed incrementally and sequentially on different partitions.
As each step of the squaring can be really complex we apply to each complex step the partitioning strategy analyzed in [12]. This approach is well suited in the squaring computation as at each step two T^{2^i} relations are composed and this operation can be seen as a particular case of a op b (see [12]). Moreover as the two terms to compose are equal the same partitioning strategy can be used. We call this kind of computation *partitioned squaring*.
Finally, experimental results show that (like usual breadth–first traversal) also iterative squaring is prone to higher complexity during intermediate steps. Then we often avoid reaching the transitive closure T^* and we stop squaring at an intermediate iteration, returning a proper subset T^k of T^*. This has the advantage of reducing the complexity of squaring while lowering traversal iterations to an acceptable amount. We call *partial computation* this approach.
Sometimes it is even necessary to use k values not being powers of 2. In such cases, considering that k can be expressed following a binary decomposition as $k = \sum_i k_i \cdot 2^i$, T^k is easily obtained using Equation (1) to compose the various T^{2^i} such that $k_i = 1$.
Once one of the previous method has been applied, traversing a FSM in a forward or in a backward fashion is thus equivalent to compute the least fixed point of the following recurrence equations:

$$\begin{aligned} \mathsf{R}_i(y) &= \mathsf{R}_{i-k}(y) + \exists_{s,x_c} (\widehat{\mathsf{T}}^k(s,x_c,y) \cdot \mathsf{R}_{i-k}(y)) \\ \mathsf{BR}_i(s) &= \mathsf{BR}_{i-k}(s) + \exists_{s,x_c} (\widehat{\mathsf{T}}^k(s,x_c,y) \cdot \mathsf{BR}_{i-k}(s)) \end{aligned}$$

When transitive closure is used ($\widehat{\mathsf{T}}^k = \widehat{\mathsf{T}}^*$) a single traversal step is enough to compute all reachable states.

4 Verification and Synthesis

In general, when circuits include counters as well as other components, neither the standard traversal nor iterative squaring may be successful. This is often the case found in hardware and software systems where counters have a complex structure (two or more registers, a free–run counter, comparators, etc.) well suited to perform different functions (see Section 5).
We propose an *ad-hoc* techniques, based on iterative squaring as described in the previous section, to solve some of the problems encountered in verification and synthesis. We will analyze these two fields separately in the next two subsections.

4.1 Verification

Verification is often based on product machine traversal. The product machine is composed by the two FSMs that evolve in lockstep, each according to its next state function, sharing the inputs and letting the outputs record the difference between the two. This technique usually increases the complexity of the problem, especially when the counters compared have different internal structures (e.g. an up–counter vs. a down counter).
Given a counter and an initial state we aim at verifying the following timing properties of *terminal count*, without resorting to the product machine model:

- Reachability, i.e. terminal count is guaranteed to occur at some time in the future.
- Time interval of occurrence, i.e. terminal count will occur within a given time interval or possibly at a given time.
- Equivalence between counters, i.e. two counters produce simultaneous terminal counts.

We limit our analysis to single terminal count intervals, but extending it to multiple intervals or periodicity is trivial. Moreover we consider, in this paragraph, the original non–deterministic transition relation T but everything can be done using the constrained relation $\widehat{\mathsf{T}}$.
All the above verification problems can be dealt with symbolically by reducing them to one of the following:

1. *Verifying* that terminal count occurs within a certain time interval, i.e. after t_1 and before t_2 clock cycles (possibly $t_2 = \infty$).
2. *Computing* the time interval, i.e. $t_1 \div t_2$, at which terminal count occurs.

Let TC represent the Boolean function related to the terminal count output.
In the first case (proving that terminal count occurs between t_1 and t_2) we prove that:

$$((\mathsf{R}_{t_1-1} \cdot TC) = 0) \cdot ((\mathsf{R}_{t_2} \cdot TC) = 1) \cdot ((\mathsf{R} \cdot \overline{\mathsf{R}_{t_2}} \cdot TC) = 0)$$

Informally, TC is 0 for all states reachable in $t_1 - 1$ steps, it is 1 for some states reachable in less than t_2 steps and it is 0 for states reachable in more than t_2 steps.
Reachable state sets R_{t_1-1}, R_{t_2} and R are easily computed using T^{t_1-1}, T^{t_2} and T^* or proper subsets, using the *partial approach* (see Section 3).
In the second case (computing the time interval $t_1 \div t_2$ in which terminal count occurs) the first occurrence of terminal count is at time t_1 iff:

$$((\mathsf{R}_{t_1-1} \cdot TC) = 0) \cdot ((\mathsf{R}_{t_1} \cdot TC) = 1)$$

t_1 is computed through a binary search: starting from the initial state set S_0, we first compute the largest i such that

$$((\mathsf{R}_{2^i} \cdot TC) = 0) \cdot ((\mathsf{R}_{2^{i+1}} \cdot TC) = 1)$$

then we repeat this search using R_{2^i} as initial state set. The process find decreasing i values and terminates as soon as $i = 0$. The sum of all 2^i's found, augmented by 1, is the t_1 value required. Again, iterative squaring is extensively adopted to allow computing the reachable state sets considered. A similar method can be used to evaluate also the last occurrence of terminal count t_2.

4.2 Synthesis

To program a counter for terminal count at time t two subproblems must be solved:

- Finding the initial state (or the set of initial states) S_0, from which terminal count is reached in t clock cycles with counter in counting mode.
- Finding a resetting (or programming) input sequence able to lead the counter to the required initial state S_0.

We avoid discussing the latter problem, because resetability is a well known problem in the field of symbolic techniques [8], [9] and it usually requires no more than a few traversal steps.
If finding the initial set is the concern, we solve it through a backward traversal using a properly squared transition relation. S_0 is in fact the set of states from which terminal count can be reached in t counting steps, but not in $t-1$ steps.
Taking $TC = 1$ as starting set for backward traversal, S_0 can thus be computed as:

$$S_0 \quad = \quad \mathsf{BR}_t \cdot \overline{\mathsf{BR}_{t-1}}$$

where iterative squaring can be used to compute T^{t-1} or proper subsets of it. As in the previous paragraph the constrained relation, $\widehat{\mathsf{T}}$, can be used as all the other techniques introduced in Section 3.

5 Experimental Results

We implemented our technique based on iterative squaring and standard traversals on top of the Colorado University Decision Diagram (CUDD) package [10]. Our experiments ran on a 200 MHz DEC Alpha with a 256 Mbyte main memory, by imposing a working memory limit of 228 Mbyte.
We experimented on a set of scalable home–made and industrial circuits that we will introduce in Subsection 5.1 whereas Subsection 5.2 will report the results obtained.

5.1 Benchmarks

Among modeled and verified scalable home–made circuits we will report experimental results on the following ones:

1. cnt a "pure" ripple–carry up–counter.
2. cnt_prog_a a programmable counter that contains a n–bit parallel to parallel register and a n–bit ripple–carry counter; the value of the register can be directly loaded from primary inputs; this value can be used to set the initial value of the counter at the beginning of each counting phase.
3. cnt_prog_b a programmable counter that contains a n–bit parallel to parallel register, a n-bit ripple–carry counter and a n–bit comparator; the counter is loaded with a value of all–zero whereas its outputs are compared with the value of the register (directly loaded from primary inputs) to produce the terminal count.

cnt is used just to prove the efficiency of iterative squaring on "pure" large counters. cnt_prog_a reaches all the states, $2^{2\,n}$, in just three steps as the initial value loaded into the register depends on primary inputs. We need a setting phase, of 2 clock cycles, to load the counter with the desired value. Similarly cnt_prog_b reaches all the states $2^{2\,n}$ in 2^n clock cycles and it also needs a setting phase to count properly. In all these examples a pedix is used to indicate the size of the counter in term of bits.
On the other hand the industrial counters that we have used originate from the output compare functions of the timing unit of the Motorola 68HC11 micro–controller [11]. The timer unit is composed of:

- A programmable clock pre–scaler (SCALE), that can divide the CPU master clock by 1, 4, 8 and 16. In all our experiments we have used the larger scaling factor.
- A 16–bit free running counter (FRC). Including the scaling factor, this counter can count up to 2^{20} clock cycles.
- A set of 16–bit registers (OCREG), whose content is continuously compared with the free running counter, to generate various sorts of events when the two are equal.

We have chosen to model and verify the following functions, implemented with the 68HC11 timer unit:

1. oc1_self a self–triggered fixed period counter that generates a terminal count every 197520 clock cycles, by scaling the clock module 16, and then counting up to 12345. It has a state space of about 2^{36} states[2], because after every terminal count, the current value of FRC plus 12345 is loaded into OCREG. This means that all the $2^{16} \times 2^{16} \times 2^4$ states of the combined state space are reachable, because 12345 and 2^{16} are relatively prime numbers.
2. oc1_fix a fixed period counter that generates a terminal count 197520 clock cycles after receiving each *start* input. It also has a state space of about 2^{36} states.
3. oc1_prog_abs a programmable absolute counter that generates a terminal count output when FRC is equal to the value loaded in OCREG. The loading of OCREG occurs every time the *start* input is received. It also has a state space of about 2^{36} states.
4. oc1_prog_rel a programmable relative counter that generates a *terminal count* output $n \times 16$ clock cycles after receiving each *start* input, where $0 < n < 2^{16}$ is the value loaded in OCREG every time the *start* input is received. It also has a state space of about 2^{36} states.

These counters are representative of the functions available also in other similar devices, such as the Intel 8254.

[2] The actual state space is somewhat larger, because it also includes "wasted" states due to the communication protocol with the outside world and between the counters.

Circuit	|T|	|T*|	# Reached States	# Partitions	$\|T^i\|_{peak}$	# Iterations	Memory	Time
oc1_self	5240	12962	$1.3750 \cdot 10^{10}$	1	33355	21	8.3	13474
				1	31336	21+2	7.8	11266
oc1_fix	3817	17552	$2.190 \cdot 10^{11}$	1	*ovf*	*ovf*	*ovf*	*ovf*
				8	71695	21	13.5	27.5^h
				2	35085	21+8+4	7.8	16291
oc1_prog_abs	1999	1691	$2.190 \cdot 10^{11}$	1	10587	22	5.4	337
				2	7940	22+4	4.5	68
oc1_prog_rel	2030	1611	$2.190 \cdot 10^{11}$	1	9871	22	4.8	258
				1	8370	22+4	4.6	62

Table 2: Improved Iterative Squaring results on industrial circuits. *ovf* means overflow on time ($> 36^h$).

5.2 Experimental evidence

Table 1 reports experiments on standard iterative squaring to prove its effectiveness on "pure" counters.
Column Circuit indicates the name of the circuit, |T| the number of nodes of the BDDs representing T, Memory is the working memory size used by the CUDD package (in Mbytes) and Time indicates the CPU time (in seconds, unless otherwise stated).
In these examples all the states are reachable (2^n for circuit cnt and $2^{2 \cdot n}$ for circuits cnt_prog_b) then the number of nodes of T^* is equal to 1 and the number of iterations of the squaring procedure is equal to n. Circuit cnt_prog_a is not presented as without a programming phase all the states are reached in just 2 clock cycles.
If one considers that the standard breadth–first traversal requires 2^n clock cycles to deal with these counters, the advantage of iterative squaring is evident.
Table 2 compares the standard iterative squaring with the improved one proposed in Section 3 on industrial circuits. The new columns are the following: $|T^*|$ indicates the number of nodes of BDDs representing T^*, # Reached States indicates the number of states of the reachable state set, # Partitions indicates the maximum number of partitions used to deal with the experiment (following [12], i.e the partitioned approach, see Section 3), $|T^i|_{peak}$ reports the number of nodes of the largest BDD representing T^i. # Iterations indicates the number of iterations performed in each of the incremental squaring phases (see Section 3); if the incremental approach is applied this column reports more than one number and each number indicates the subsequent number of steps of one iterative squaring phase.

Circuit	|T|	Memory	Time
cnt_{64}	316	1.6	4
cnt_{128}	636	5.5	30
cnt_{256}	1276	9.3	193
$cnt\_prog\_b_{64}$	1070	9.4	77
$cnt\_prog\_b_{128}$	2537	20.9	897
$cnt\_prog\_b_{256}$	5097	63.6	8210

Table 1: Standard Iterative Squaring results on home made circuits.

Tuning the package is important. For example, in the case of circuit oc1_fix the transitive closure T^* is not directly computable (first row). Anyway we succeeded in squaring it, using the partitioned approach (second row), by taking 8 partitions of the squaring but this still required more than 27^h. Using the incremental approach, with three steps of iterative squaring with depth 21, 8 and 4, together with the partitioned one with 2 partitions, reduced the peak of the BDDs and the CPU time to 16291 seconds.
The partial approach (stopping squaring before the transitive closure) reduces the CPU time even more in some experiments; for example circuit oc1_fix can be traversed in 3291 seconds if we do just 13 steps of squaring and resort to 256 steps of traversal. Analogously, the time for circuit oc1_self can be reduced to 7890 seconds.

Circuit	N	|T|	$\|T^N\|$	Mem.	Time
$cnt\_prog\_a_{64}$	147	1021	4635	3.6	7
	114799	6223	5138	6.9	32
$cnt\_prog\_b_{64}$	799	750	2047	3.9	9
	954321	2045	2176	5.3	38
oc1_prog_abs	7	1999	4389	4.6	7
	197520	1999	1691	5.4	273
	8387608	1999	1691	5.5	341
oc1_prog_rel	99	2030	6110	4.8	19
	197520	2030	1611	4.9	196
	8000003	2030	1611	5.2	309

Table 3: Results on Verification by Iterative Squaring.

Verification (see Sections 3 and 4.1) is performed using: 1) a few steps of standard breadth–first traversal (if necessary to fix initial values) and 2) iterative squaring in the forward fashion (for the counting phase) with the constrained approach (if necessary to fix particular control input). The first phase is inexpensive. The second one has a complexity that depends on the complexity of the constrained iterative squaring phase (up to one order of magnitude simpler that corresponding "unconstrained" squaring phase) and on other typical operations of verification. For example, for circuit cnt_prog_a an initial value is loaded into the parallel to parallel register in the first clock cycles and then the loading function is disabled. This simplify the transition relation and then the iterative squaring. On the other

hand verifying when the terminal count is equal to one requires the use of the dichotomical approach and testing on TC and this require a little time.
Table 3 reports experiments on verification using this technique. All the experiments strongly benefit from input constraining adopted to separate different operating modes while squaring. Column N indicates that we program the counter to give a terminal count after N counting steps and then we try to verify this property on the counter.
For each circuit a few experiments are reported using different value of N.
Synthesis (see Sections 3 and 4.2) is performed similarly: 1) iterative squaring in the backward fashion is used to force a counting phase of N clock cycles and (if necessary, with the constrained approach to fix particular control input) and 2) a resetting sequence is found.

Circuit	N	|T|	$\|T^N\|$	Mem.	Time
oc1_prog_abs	7	1999	4389	4.6	9
	197520	1999	1691	5.4	347
	8387608	1999	1691	5.5	414
oc1_prog_rel	99	2030	6110	4.8	28
	197520	2030	1611	4.9	266
	8000003	2030	1611	5.2	401

Table 4: Results on Synthesis by Iterative Squaring.

Table 4 reports experiments on synthesis using this approach. Column N indicates that we want to program the counter with a terminal count occurring after N counting steps and we want to establish how to set the counter. For each circuit we did three experiment varying the value of N, using a small, a medium and a large value.

6 Conclusions

Symbolic FSM state space exploration techniques represent one of the major recent results of formal verification. One of their limit resides in the inability to deal with very depth circuit. Iterative squaring on the other hand is well suited to deal with "pure" counters with a very high depth. In this paper we propose an extension of the iterative squaring to general structures containing counters. The methodology is proved to be effective both in verification and synthesis.
Future work will consist in analyzing much more general circuits, e.g. systems found in *hardware and software co–design* methodology and in micro–controllers. Moreover we are currently extending the methodology with a new approach based on a *disjunctive partitioned transition relation* and on iterative squaring. The disjunctive partitioned approach to transition relation allows us to represent and visit proper subsets, i.e. with a compact BDD representation, of the state transition graph. These subsets can have very long sequential depth and iterative squaring can help to visit and to "compress" them.

References

[1] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni–Vincentelli, "Hardware/software co–design of embedded systems," *IEEE Micro*, August 1994, Vol. 14(4), pp. 26–36

[2] S. Campos, E. Clarke, W. Marrero, M. Minea, "Verus: a tool for quantitative analysis of finite-state real-time systems," in *Proc. ACM SIGPLAN'95 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1995

[3] R. Hojati, H. Touati, R.P. Kurshan, R.K. Brayton, "Efficient omega–regular language containment," in *Proc. CAV'94*, June 1994

[4] E. Macii, B. Plessier, F. Somenzi, "Verification of systems containing counters," in *Proc. IEEE ICCAD'92*, November 1992, pp. 179–182

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," in *Proc. ACM/IEEE DAC'90*, June 1990, pp. 46–51

[6] Y. Matsunaga, P.C. McGeer, R.K. Brayton, "On computing the Transitive Closure of a State Transition Relation," in *Proc. ACM/IEEE DAC'93*, June 1993, pp. 260–265

[7] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Transactions on CAD*, Vol. 13, No. 4, April 1994, pp. 401–424

[8] C. Pixley, "A computational theory and implementation of sequential hardware equivalence," in *AMS/DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 3, 1991, pp. 293–320

[9] C. Pixley, S.-W. Jeong, G.D. Hachtel, "Exact Calculation of Synchronization Sequences Based on Binary Decision Diagrams," in *Proc. ACM/IEEE DAC'92*, June 1992, pp. 620–623

[10] F. Somenzi, "CUDD: CU Decision Diagram Package – Release 1.0.4," Technical Report, Dept. of Electrical and Computer Engineering, University of California, Boulder, November 1995

[11] *M68HC11 Reference Manual*, Motorola inc., 1991

[12] G. Cabodi, P. Camurati, S. Quer, "Improved Reachability Analysis of Large Finite State Machine," in *Proc. IEEE/ACM ICCAD'96*, November 1996, pp. 354–360