Timed Circuit Synthesis Using Implicit Methods *

Robert A. Thacker Wendy Belluomini
Computer Science Department
University of Utah
Salt Lake City, UT 84112

Chris J. Myers
Electrical Engineering Department
University of Utah
Salt Lake City, UT 84112

Abstract

The design and synthesis of asynchronous circuits is gaining importance in both the industrial and academic worlds. Timed circuits are a class of asynchronous circuits that incorporate explicit timing information in the specification. This information is used throughout the synthesis procedure to optimize the design. In order to synthesize a timed circuit, it is necessary to explore the timed state space of the specification. The memory required to store the timed state space of a complex specification can be prohibitive for large designs when explicit representation methods are used. This paper describes the application of BDDs and MTBDDs to the representation of timed state spaces and the synthesis of timed circuits. These implicit techniques significantly improve the memory efficiency of timed state space exploration and allow more complex designs to be synthesized. Implicit methods also allow the derivation of solution spaces containing all valid solutions to the synthesis problem facilitating subsequent optimization and technology mapping steps.

1. Introduction

Recent trends in the integrated circuit industry, such as decreasing feature sizes and increasing clock speeds, make global synchronization across large chips difficult to maintain. As a result, many designers have become interested in asynchronous circuits because they eliminate the need for global synchronization. Asynchronous circuits consist of groups of independent modules which communicate using handshaking protocols. Since there is no global clock, clock distribution and skew are not issues. Also, eliminating the global clock permits modules to work at their own pace and allows average-case performance to be realized. There are a number of different styles for designing asynchronous circuits. Most asynchronous design methodologies are based on the assumption that nothing is known about the delays between signal transitions. Therefore, the circuit must be constrained to work correctly even in cases which never occur in a realistic implementation. The overhead necessary to guarantee this behavior often makes the asynchronous average-case performance worse than the synchronous worst-case.

Timed circuits are a class of asynchronous circuits which use explicit timing information in circuit synthesis. Precise timing relationships are often unknown before synthesis and technology mapping. However, applying even rough estimates can lead to the removal of large amounts of circuitry that would be required for a speed-independent design. These timing assumptions can then be formally verified after synthesis when the actual timing values are known. This design style can lead to significant gains in circuit performance over asynchronous circuits designed without timing assumptions [15].

The first stage of timed circuit synthesis involves the exploration of the timed state space to determine which untimed states are reachable by the system. The circuit is first specified using a formalism that allows a lower and an upper bound to be assigned to the causal relationships between signals. Timing analysis is then performed by our design tool `ATACS` using geometric regions and partially ordered sets (POSETS) of events, which has been shown to be an efficient method for representing information about timed state spaces [3, 4, 16, 17]. We use *Binary Decision Diagrams* (BDDs) [7] and *Multi-terminal Binary Decision Diagrams* [10] to efficiently represent these timed state spaces.

The second stage of synthesis consists of repeatedly dividing the state graph into subregions to determine the necessary behaviors. For each signal, the graph is divided into those regions where the signal should be enabled to rise, should be enabled to fall, should remain high, or should remain low. Equations are derived to represent a circuit implementation which conforms to these behaviors. Our synthesis method uses BDDs allowing the derivation of solution spaces containing all valid solutions to the synthesis problem.

*This research is supported by NSF CAREER award MIP-9625014, SRC contract 97-DJ-487, an SRC Graduate Fellowship, and Intel Corp.

2. Timed state space exploration

Timed circuit synthesis is dependent on a complete exploration of the timed state space of the specification. This state space can be very large since it must include, not only all of the combinations of signal values allowed by the specification, but also the time relationships between signal firings. However, it can be smaller than the complete state space of an equivalent specification without timing since states that are not reachable given the timing information are not explored.

The size of the timing information depends on the timing algorithm being used. One representation is to attach a clock to each signal transition that advances only in discrete time steps [8]. This representation can cause state space explosion, especially for large delay ranges [17]. A BDD method has been proposed in [6], to improve discrete time memory performance, but it does not address the state explosion problem inherent in discrete time. The geometric region method, where timing information is stored as a constraint matrix representing relationships between signal transition times, has been shown to be an efficient way to represent a timed state space [3, 14, 16, 17]. However, even with a region based representation, the memory required to store such a state space explicitly can be prohibitive for large designs. In many domains, implicit methods have been shown to significantly reduce memory usage [7]. Since timed state space exploration is such a memory intensive process, it is an excellent candidate for such an approach.

2.1. Motivating example

The block diagram in Figure 1 is for a controller for a self-timed FIFO. In [13], a highly optimized timed circuit implementation is presented which is designed by hand. The correctness of this circuit is highly dependent on timing parameters. This example is used to show how our timed circuit synthesis method can derive the same efficient circuit from the description of the required behavior and the known timing parameters. The basic behavior is that when a request comes in (i.e., $FIN+$) and the fifo is empty (i.e., $EOUT$ is high), the data is latched (i.e., En_bar+ and $En-$). In parallel, the insertion is acknowledged (i.e., $SEOUT\_-$) and the next stage is requested to accept the data (i.e., $FOUT+$). When the next stage accepts the data (i.e., $SEIN\_-$), the FIFO is set to be empty (i.e., $EOUT+$) and the latch is opened (i.e., En_bar- and $En+$).

2.2. Explicit timed state space exploration

The state space exploration procedure used by ATACS begins with a *timed event/level (TEL) structure*, described

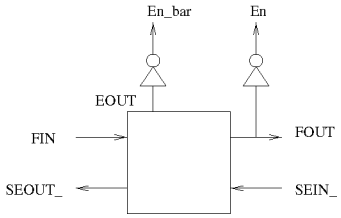

Figure 1. Block diagram for a timed FIFO.

formally in [2]. TEL structures can represent a set of specifications equivalent to those represented by both time and timed Petri nets, as well as others that are quite difficult to represent with a Petri net. A TEL structure consists of a set of rules that represent causality between signal transitions, or *events*, as well as a set of conflicts which are used to model disjunctive and choice behavior. Each rule is of the form $\langle e, f, l, u, b \rangle$, where e is the *enabling event*, f is the *enabled event*, $\langle l, u \rangle$ is the bounded timing constraint, and b is a boolean expression which must be satisfied before f is allowed to occur. A rule is said to be *enabled* if its enabling event has occurred and its boolean expression evaluates to true. The timing constraint places a lower and upper bound on the timing of a rule. A rule is *satisfied* if the amount of time which has passed since the enabling event has exceeded the lower bound of the rule. A rule is said to be *expired* if the amount of time which has passed since the enabling event has exceeded the upper bound of the rule. Ignoring conflict, an event cannot occur until *all* rules enabling it are satisfied. An event must always occur before *every* rule enabling it has expired. Since an event may be enabled by multiple rules, it is possible that the differences in time between the enabled event and some enabling events exceed the upper bound of their timing constraints, but not for all enabling events.

A graphical representation of the TEL structure for the FIFO example is shown in Figure 2. Nodes represent signal transitions and arcs represent causal relationships between them. Tokens on arcs indicate that the preceding transition has occurred but the following transition has not. All incoming arcs must have tokens to fire a transition.

The goal of state space exploration is to derive the *state*

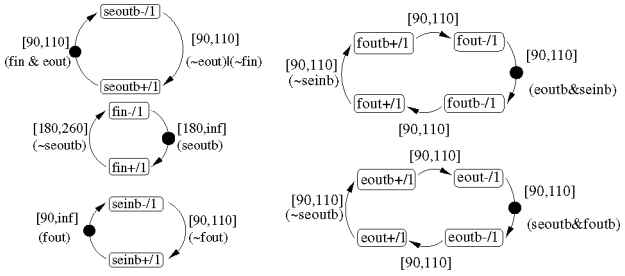

Figure 2. The TEL structure for the FIFO.

graph (SG), which is necessary for circuit synthesis. A SG is a graph in which the vertices are untimed states and the edges are possible *state transitions*. A transition between two states exists if the specification allows the circuit to move from one state to the other with one signal transition. A *reduced state graph* (RSG) is a SG where some branches have been pruned because timing information has shown them to be unreachable. A RSG is modeled by the tuple $\langle I, O, \Phi, \Gamma \rangle$ where I is the set of input signals, O is the set of output signals, Φ is the set of states, and $\Gamma \subseteq \Phi \times \Phi$ is the set of edges. For each state s, there is a corresponding labeling function $s : I \cup O \rightarrow \{0, R, 1, F\}$ which returns the value of each signal and whether it is enabled (i.e., 0 if x is stable low, R if x is enabled to rise, 1 if x is stable high, F if x is enabled to fall). A state transition (s, s') is often denoted as follows: $s \xrightarrow{x} s'$ where x is the signal that changed value.

Since our TEL structure specifications include timing constraints, it is necessary to use a timed state space exploration algorithm to find the reachable state space. The method used is to perform a depth first search to find all reachable timed states. A timed state for a TEL structure consists of a set of rules whose enabling events have fired, R_m, the state of all the signals, s_c, and a set of timing information, TI. The timing information, TI, is represented with geometric regions, which were first introduced in [5, 11].

When the geometric region approach is used for timing analysis, a constraint matrix M specifies the maximum difference in time between the enabling times of all the currently enabled rules. The $0th$ row and column of the matrix contain the separations between the enabling times of each enabled rule and a dummy rule $r_\emptyset$. The enabling time of $r_\emptyset$ is defined to be uniquely 0. Each entry m_{ij} in the matrix M has the value $max(t(enabling(j)) - t(enabling(i)))$, which is the maximum time difference between the enabling time of rule j and the enabling time of rule i. Since the enabling time of $r_\emptyset$ is always zero, the maximum time difference between the enabling of rule i and the enabling of rule $r_\emptyset$ (m_{0i}) is just the maximum time since i was enabled. The maximum time difference between the enabling time of $r_\emptyset$ and the enabling time of rule i (m_{i0}) is the negation of the minimum time since i was enabled. Note that M only needs to contain information on the timing of the rules that are currently enabled, not on the whole set of rules. Figure 3(a) shows a sample geometric region, and Figure 3(b) shows the corresponding constraint matrix. The region is a convex polygon defining the relationships between the timers associated with the active rules at a given point in the state space exploration, and the matrix is a concise numerical description of the region. In this case, the region indicates that the timer t_1, associated with rule r_1, can have a value anywhere from two to twenty time units, but no more than five time units greater than t_2. ($t_0 - t_1 \leq -2, t_1 - t_0 \leq 20$, and $t_1 - t_2 \leq 5$.) Similarly, timer t_2, associated with rule r_2 can have a value between zero and fifteen, but must be no more than two time units less than t_1. ($t_0 - t_2 \leq 0, t_2 - t_0 \leq 15$, and $t_2 - t_1 \leq -2$.) The polygon shown in Figure 3(a) contains all points (t_2, t_1) which conform to these timing constraints.

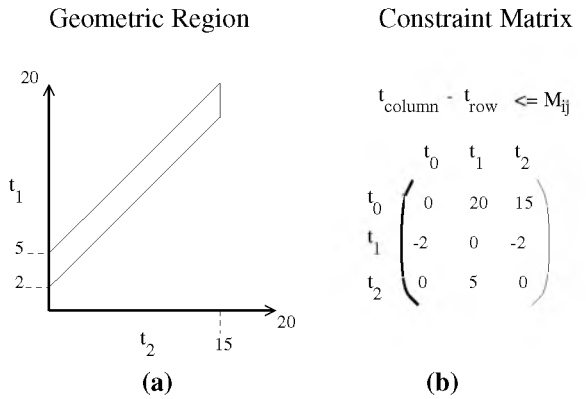

Figure 3. A geometric region and matrix.

2.3. Implicit timed state space exploration

The explicit enumeration method described above requires too much memory to effectively represent complex designs. Therefore, it is necessary to explore alternative methods of storing this information. Since much of the data compiled during state space exploration consists of simple bit vectors, we have chosen to use BDDs, which have been shown to be a highly efficient method for storing and manipulating Boolean functions [7]. Because geometric region information is integer-valued, MTBDDs have been chosen to store the region matrices. MTBDDs are a type of BDD which allow terminal nodes to contain numerical data, rather than just the constants TRUE and FALSE. Geometric region matrices only have entries for currently enabled rules. However, to make the representation more manageable, the matrices are expanded to a canonical form, where rows and columns representing rules that are not enabled have been filled with a "not an entry" symbol, the constant FALSE. MTBDDs collapse paths with common structural features to the fewest nodes possible. In addition, because of the nature of BDD implementations, it is possible for separate geometric regions with similar structures to have common subregions stored in the same memory location.

The MTBDD for a region is composed of three parts. First, a BDD is constructed to store the bit vector that indicates which rules are in R_m. Second, it is necessary to store the list of regions associated with each R_m set. To represent this list structure, a numerical index i is used to indicate that a given matrix is the i^{th} matrix associated with a given R_m

set. Any number i can be viewed as a bit vector and represented as a BDD. In order to conserve space, precisely enough bits are used to represent the largest number currently needed. Finally, the geometric region matrix must be represented. There are many ways to represent regions, and on the surface using MTBDDs would seem to be a very inefficient method. Methods for representing sparse matrices have been developed for scientific computing that are much more efficient at representing single matrices. There are two major benefits to this approach. First, it allows matrices to be manipulated within the BDD paradigm, which among other advantages allows two matrices to be compared for equality in constant time regardless of size. The greatest advantage, however, is the capacity to amortize the costs of storage across many matrices. Many of the matrices encountered in practice differ very little from one another. The BDD storage system used in ATACS allows additional matrices to be added to the database and only consume the resources necessary to represent the new elements. This often leads to the use of only a few BDD nodes per matrix.

A matrix with integer entries can be viewed as a function ($N \times N \mapsto Z$). With row and column indices parameterized as binary numbers, this function becomes $\{0,1\}^n \times \{0,1\}^n \mapsto Z$. The geometric region matrices are thus parameterized and stored as a function which takes row and column indices and returns the appropriate matrix entry ($M(r,c) = M_{rc}$). MTBDDs are an ideal way to represent this type of function [10]. Figure 4 shows the complete MTBDD for the timed state where $R_m = \{r_1, r_3\}$, the link value is 2, and the region is the one shown in Figure 3. When a new timed state is found, the timed state list MTBDD T_S is extended by the call: $T_S = ITE(FindR_mBDD(R, R_m, \mathbf{m}) \wedge i, MakeMatrixBDD(n, M, \mathbf{r}, \mathbf{c}), T_S)$. In this formula, $FindR_mBDD$ constructs a BDD for the R_m set, $MakeMatrixBDD$ constructs a MTBDD for the region, i is the list index BDD for this region, and ITE is the *if-then-else* operator.

To represent the reachable state space, a predicate S on the vector $\mathbf{x}$ is defined which returns true for all states reachable in any number of transitions from the initial state. The vector $\mathbf{x}$ is $\langle x_1, x_2, ...x_n \rangle$, where each variable x_i is in $I \cup O$.

The NextState function N is a predicate on $S \times S$ which returns true for all the state pairs (s, s') for which s' may be reached from s in exactly one signal transition. A complication arises from the use of timing in generating the RSGs. As mentioned before, when timing considerations show a state to be unreachable, it may be removed from the RSG. If we based our implementation only on the reduced state graph, the enablings to reach these states would be lost, and the resulting circuit would be suboptimal. In the FIFO example, a naive derivation of S and N results in the circuit found in Figure 5(a) for *SEOUT_*. This *generalized C-element* circuit may work, but it is a larger and slower gate than the simple NAND gate shown in Figure 5(b) which is derived from the correct RSG. This problem is solved in the explicit system by using a four valued logic system (0, R, 1, and F as described above). However, in the implicit method the use of bit vectors makes this less attractive, as it would double the necessary length of the vectors.

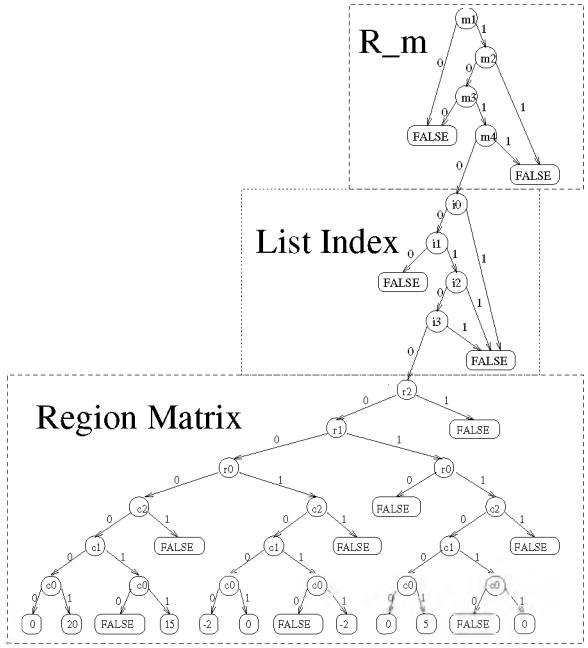

Figure 4. MTBDD for a timed state.

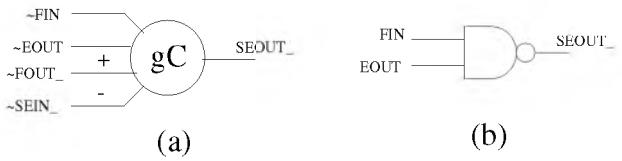

Figure 5. Gates for *SEOUT_*.

The basic problem can be illustrated using the familiar diamond shown in Figure 6. The original speed-independent graph is shown in Figure 6(a). For the timing specified in the TEL structure in Figure 6(b), the signal b always rises before a, so the state (1R) is removed from the graph. If the correct enablings are not maintained, the less concurrent graph shown in Figure 6(c) is produced. The enabling of a is now delayed by the time necessary to fire b, and each cycle of the circuit is slowed by that amount. In other words, the improperly pruned graph loses the fact that x was the enabling event, and actually represents the TEL fragment in Figure 6(d). The total time necessary to traverse this graph from state (RR) to state (11) should be 10 time units, but instead it increases to 15 time units. This less concurrent circuit may not only be slower, but it may also

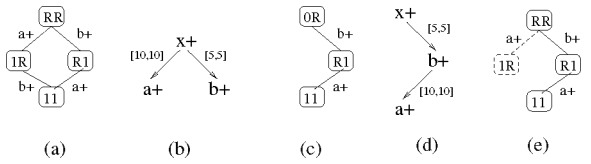

Figure 6. "Ghost state" example.

be incorrect if it violates the original timing assumptions.

To maintain the correct enablings, the N relation is populated with a transition for every enabled signal, even if the target state is not reachable. Such a "ghost transition" can be detected by the fact that the target state is not contained in the S relation. This ghost state consists of the same values as the original state, except that the enabled signal has changed phase.

Figure 6(e) shows an example of a "haunted" graph: the state (1R) has been reinserted as a "ghost state" with a transition from (RR). This path is never taken, but it is essential that it be represented.

2.4. Results

We have implemented the implicit timed state space exploration procedure and tested it on a number of examples. Since most timed circuit examples are quite small due to previous memory limitations of synthesis tools, we have parameterized our asynchronous FIFO example in order to demonstrate the effectiveness of implicit methods. This FIFO is very concurrent when parameterized and generates an extremely large number of geometric regions which correspond to the number of regions necessary to synthesize a large complex design. The POSET timing method for state exploration discussed in [3] is used to generate the timed state space. The examples shown were run on a 400 MHz PentiumII with 384Mb of memory.

Figure 7 shows the memory usage pattern of the state space exploration for 4 stages of the timed FIFO for both the explicit and implicit methods. The x-axis shows the number of regions explored and the y-axis shows the maximum memory used to that point in the state space exploration. The solid lines represent the implicit method and the dashed lines represent the explicit method. The graphs show that the implicit method not only yields a significant overall improvement in memory usage, but also that the memory usage trends for the implicit method are much better. As the number of regions grows very large, the amount of memory used by the implicit method approaches an asymptotic value. This occurs since once the BDDs get mostly full, adding additional regions does not add significant memory due to the node sharing behavior of BDDs. When the BDDs get large and a new region is added, most of the nodes needed for this state are already in the current BDD, and very little new memory is necessary. With explicit methods, on the other hand, each new region throughout the state space exploration requires a new allocation of memory, causing the memory usage of the explicit method to grow linearly with the number of regions.

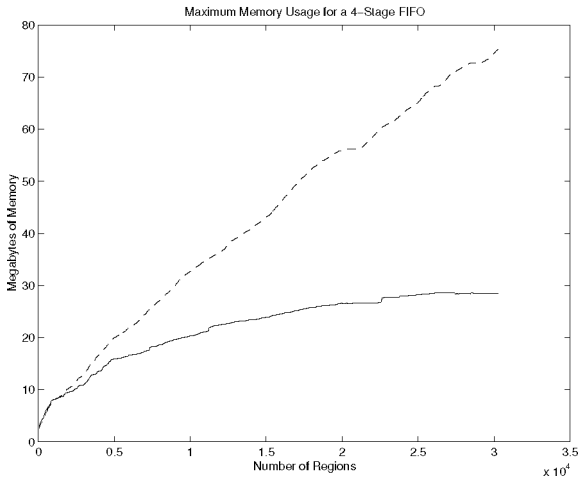

Figure 7. FIFO memory performance.

3. Synthesis

The synthesis stage starts with a *reduced state graph* (RSG) implicitly stored as described in the previous section using two BDD structures: S, the reachable state space, and N, the next state relation.

3.1. Excitation regions and quiescent states

In order to obtain an implementation, the state space is first decomposed for each output signal into a collection of *excitation regions*. An excitation region for the output signal x is a maximally connected set of states in which the signal is enabled to change to a given value (i.e., $s(x) = R$ or $s(x) = F$). If the signal is rising in the region (i.e., $s(x) = R$), it is called a *set region*, otherwise the region is called a *reset region*. The excitation regions for each signal transition are indexed with the variable k and the k^{th} excitation region for a signal transition $x*$ is denoted $ER(x*, k)$, where "*"indicates "↑" for set regions and "↓" for reset regions. We also define a set of *excited states* $ES(x*)$, which is the union of the excitation regions for a given signal transition, i.e., $ES(x*) = \bigcup_k ER(x*, k)$.

For each signal transition, there is an associated set of stable, or *quiescent*, states $QS(x*)$. For a rising transition $x\uparrow$, it is the states where the signal is stable high (i.e., $QS(x\uparrow) = \{s \in \Phi \mid s(x) = 1\}$), and for a falling transition, it is the states where the signal is stable low, i.e., $QS(x\downarrow) = \{s \in \Phi \mid s(x) = 0\}$).

Given the BDD N, the BDD representations of ES and QS are straightforward to find. For instance, the set of excited states for $x\uparrow$ would be found by applying the following formula: $ES(x\uparrow) = exist_q(\mathbf{x}', \neg x \wedge x' \wedge N)$. And the quiescent states can be found in a similar manner: $QS(x\uparrow) = x \wedge S \wedge \neg ES(x\uparrow)$.

The function $exist_q(x, f)$ is defined to be the existential quantifier of the variable x in the function f. This is equivalent to $f_x \vee f_{\neg x}$, and is used to return the portion of the predicate which can return *TRUE* for any value of x. This function is extended to iteratively operate on a vector of variables $\mathbf{x}$, and results in a new function f' which does not depend on the variables in $\mathbf{x}$.

The excitation regions would then be found by dividing each excited set into connected regions. To do this, the algorithm merely picks a seed state at random and iteratively adds all excited states reachable in one step from the region.

3.2. Timed circuit implementation

The circuit is implemented by creating a function block for each output signal, consisting of a C-element with a *sum-of-products* (SOP) stack each for the set and reset (see Figure 8). Each product block in the SOPs for each function implements a cover for a single excitation region. Note that while depicted as a simple AND gate, in order to guarantee hazard-freedom, this "product" block may need to be a more general function block. The circuit may be implemented using a standard C-element (SC) structure using discrete gates, as shown in Figure 8(a). It may also be created using a complex gate known as a *generalized C-element* (gC) [12]. Figure 8(b) shows a transistor-level gC design using a weak feedback staticizer, and Figure 8(c) shows a fully static design. If a gC only has one pullup and one pulldown stack, it can be depicted as shown in Figure 5(a). Note that a signal labelled with a "+" is only used in the pullup stack, and a signal labelled with a "-" is only used in the pulldown stack.

In [9], a parametrized family of decompositions of high-fanin gates is investigated at one time by adding additional

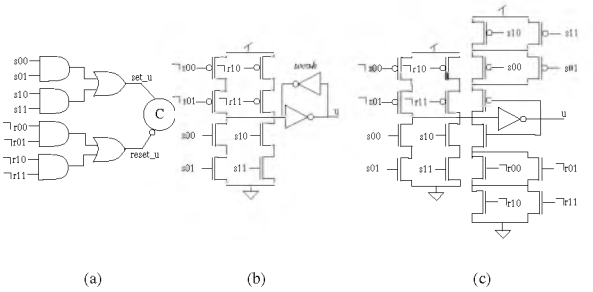

Figure 8. C-element circuit types.

variables. We extend this idea to synthesis by representing our covers by a series of implications of the form $(\chi_i \Rightarrow x_i) \wedge (\chi_{n+i} \Rightarrow \neg x_i)$. These implications are ANDed together to produce a BDD which represents every possible potential single cube cover of the corresponding ER: $C_0(x*, k) = \bigwedge_i \Psi_{i,0}$, where $\Psi_{i,0} = [(\chi_{i,0} \Rightarrow x_i) \wedge (\chi_{n+i,0} \Rightarrow \neg x_i)]$. We then apply restriction operators to this BDD, to remove covered states which violate our requirements for a valid cover. Any satisfying assignment of the remaining BDD is a valid implementation: if a χ variable appears in the positive phase, the implied variable must appear in the cover; if it appears in the negative phase, the variable cannot be included; and if it does not appear at all, it may or may not be used, at the designers discretion.

Occasionally an excitation region is found which cannot be covered by a single cube. Our algorithm creates a SOP block to represent this region, instead of a simple "AND" function. To accomplish this, the algorithm tests each cover BDD to see if it is identically FALSE. If this occurs, a second (or third, etc.) initial cover is created, and ORed together with the preceding initial cover (i.e., $C = C_0 \vee C_1 \vee ... C_m$). The resulting BDD is passed through the same filters, producing a multicube implementation.

3.3. Correct cover formulation

In order to create a valid timed circuit implementation, it is necessary to define the states a cover must include, may include, and may not include. Each cube of the implementation must include the entire corresponding excitation region. In order to minimize the logic, it may also include any unreachable state, and may include some additional reachable states. Inclusion of some reachable states, however, can cause incorrect behavior. These disallowed states vary, depending on the type of circuit chosen. In a gC implementation, any state where the signal is enabled in the same direction or stable at the final value may be included. In a SC circuit, some of those states may need to be excluded to guarantee hazard-freedom. The correctness constraints discussed here were developed in [1] for speed-independent circuits and extended to timed circuits in [14].

In a gC implementation, the allowed growth regions include the remainder of the excitation space and the entire quiescent space for the corresponding signal transition. In other words, correct covers must satisfy the following *covering constraint*: $ER \subseteq C \cap \Phi \subseteq ES \cup QS$. The boolean equation for this restriction is the following: $V = S \wedge \neg ES \wedge \neg QS$. That is, the cover may not include any reachable state not in the quiescent or excited spaces. This prevents the gate from being pulled up and down simultaneously.

In a SC implementation, additional internal signals are introduced by the use of discrete gates. In order to pre-

vent the introduction of hazards, additional restrictions are placed on the states allowed in the cover. The purpose is to ensure that each cover makes a single monotonic transition when it is actively changing the output and makes no other transitions at any other time. To guarantee this, we need a modified *covering constraint* and an *entrance constraint*. This ensures that the transition of the gate is acknowledged. The covering constraint is the following: $ER \subseteq C \cap \Phi \subseteq ER \cup QS$. In other words, the cover must include the entire ER, and may *only* include states from the ER or the corresponding QS. The resulting boolean equation is: $V_1 = S \wedge \neg ER \wedge \neg QS$. This ensures that only one AND block is on at a time, so the transition can be acknowledged by a transition on the output. In addition, the cover may only be entered through the excitation region. This is to guarantee a single monotonic transition, with no unacknowledged glitch in the function block. The entrance constraint is $((s, s') \in N) \wedge (s \notin C) \wedge (s' \in C) \Rightarrow (s' \in ER)$, and the resulting boolean equation is $V_2 = TRANS(\mathbf{x}' \rightarrow \mathbf{x}, exist_q((\mathbf{x}, N(\mathbf{x}, \mathbf{x}') \wedge \neg C(\mathbf{x}) \wedge C(\mathbf{x}') \wedge \neg ER(\mathbf{x}')))$. The final boolean equation for the violations is: $V = V_1 \vee V_2$.

The valid cover BDD, VC, is constructed to include all implementations that do not include any violating states and completely cover the corresponding excitation region. In other words, we filter the cover BDD C with the following conditions: (1) $C \cap V = \emptyset$, and (2) $C \cap ER = ER$. The combined boolean equation is $VC = univ_q(\mathbf{x}, (\neg C \vee \neg V) \wedge (\neg ER \vee C))$. The function $univ_q(x, VC)$ implements the universal quantifier. This is equivalent to $f_x \wedge f_{\neg x}$, and returns the portion of the predicate that is independent of the value of x. This can be extended to iteratively operate on the vector **x**. The resulting BDD represents all valid covers of the signal.

Figure 9 shows two possible timed circuit implementations for the FIFO controller. The circuit shown in Figure 9(a) is the one found using explicit logic synthesis. While the first circuit is also found during implicit logic synthesis, the circuit shown in Figure 9(b) is selected as it uses a simple NAND gate rather than a generalized C-element to implement *EOUT*. Both of these gates have the same cost (2 literals for the reset region), so either may be selected arbitrarily by the explicit method. In fact, we are lucky to have only one generalized C-element as *SEOUT_* and *FOUT* also have equal cost generalized C-element implementations. We do not know until after the logic optimization step whether the gate can be reduced to a combinational gate, so ending up with a combinational gate is simply a matter of luck. However, the implicit technique efficiently keeps track of all possible implementations allowing the technology mapping step to pick the one that leads to the best optimized circuit implementation. One last interesting note is the circuit shown in Figure 9(b) is exactly the one found by hand in [13].

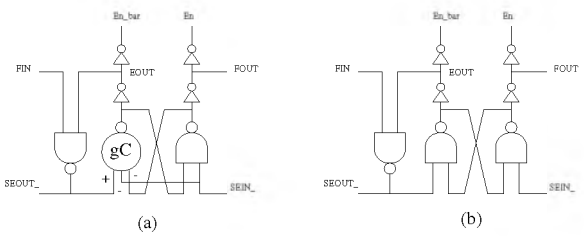

Figure 9. Timed FIFO Circuit.

3.4. Results

The complete BDD timed circuit synthesis procedure has been automated within the CAD tool ATACS. This algorithm has been applied to the design of numerous timed circuit designs. Synthesis results are shown in Table 1. The first column shows the number of gC style solutions found using the BDD synthesis procedure that can be implemented within the stack size limits. (A set of simple filters have been employed to extract the set of solutions having reasonable implementations in CMOS technology.) Most of the examples have a huge number of possible implementations with four or fewer transistors in each stack ("need decomp" is used to indicate that there is no valid implementation using only four-stacks). However, since they are stored implicitly, keeping track of this many solutions is not difficult and is useful for technology mapping. The second column shows the number of solutions for each example where each transistor stack has its minimum size. Some of these have only one minimal solution, but several have multiple minimal solutions which are not found if explicit synthesis methods are used.

The numbers in this table represent the number of potential implementations for the entire circuit. This number is the product of the possible covers for each individual excitation region. For example, in the gC implementation of the SPDOR, the set region for x has 8 solutions, the reset region has 2, each of the two set regions for a has 8 solutions, and the reset region has 8 which makes a total of $8 \times 2 \times 8 \times 8 \times 8 = 8192$. The SC implementation is more restricted so it only has 80 possible solutions. The use of implicit methods not only improves memory performance for large specifications, they also allow a parameterized family of solutions to be produced. Possibilities for component sharing between functions are also increased by the capacity to consider all valid solutions in parallel.

4. Conclusions and future work

This paper presents a design tool for the synthesis of timed circuits. This tool utilizes BDD based algorithms and data structures to allow the synthesis of larger timed circuit implementations. We formulated a MTBDD representation

Table 1. Experimental results.

	# of Solutions	
Examples	< 4	min
spdor	8192	1
spdand	512	1
cnt	614656	1
mmuoptSV	1.3×10^{23}	405
mmuopt	3.7×10^{9}	4
slatch	1.3×10^{15}	2
elatch	9.4×10^{12}	4
SELopt	need decomp	4
tsbm	need decomp	4
scsiSVT	3.2×10^{9}	18
lapb	16384	1
lapb2	1.2×10^{9}	1
lapb3	6.1×10^{15}	2
lapb4	$2{,}2 \times 10^{22}$	4
fifo	1.7×10^{11}	4
fifo2	1.9×10^{27}	16
fifo3	2.1×10^{43}	64

for the timed state spaces during timed state space exploration. We also described a BDD representation of the reduced state graph which is derived alongside. We use ghost transitions to preserve accurate signal enabling information. We have developed BDD formulations and algorithms for both standard-C and generalized C-element implementation styles. These algorithms find all valid covers for each excitation region (if necessary, by transparently finding minimal multicube covers).

The two major advantages of the implicit synthesis method is that larger timed systems can be designed and a parameterized family of solutions is found while earlier algorithms merely found a single solution. Considering all possible valid implementations will greatly facilitate technology mapping. In the future, we plan to extend BDD based technology mapping algorithms for speed-independent circuits to timed circuits.

References

[1] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng. Automatic synthesis of gate-level speed-independent circuits. Technical Report CSL-TR-94-648, Stanford University, Novermber 1994.

[2] W. Belluomini and C. J. Myers. Timed event/level structures. In collection of papers from TAU'97.

[3] W. Belluomini and C. J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[4] W. Belluomini and C.J. Myers. Verification of timed systems using posets. In *International Conference on Computer Aided Verification*. Springer-Verlag, 1998.

[5] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3), March 1991.

[6] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. International Conference on Computer Aided Verification*, 1997.

[7] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *ICCAD*. IEEE Computer Society Press, 1995.

[8] J. R. Burch. Modeling timing assumptions with trace theory. In *ICCD*, 1989.

[9] S. M. Burns. General condition for the decomposition of state holding elements. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[10] E. Clarke, M. Fujita, and X. Zhao. Application of multi-terminal binary decision diagrams. Technical Report CMU-CS-95-160, Carnegie-Mellon University, 1995.

[11] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, 1989.

[12] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive VLSI circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1990.

[13] Charles E. Molnar, Ian W. Jones, Bill Coates, and Jon Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[14] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis of gate-level timed circuits with choice. In *Proc. 16th Conf. on Advanced Research in VLSI*, pages 42–58. IEEE Computer Society Press, 1995.

[15] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[16] T. G. Rokicki. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.

[17] T. G. Rokicki and C. J. Myers. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.