ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2713158

Implicit Methods For Timed Circuit Synthesis

Article · September 1998

Source: CiteSeer

CITATIONS

7

READS

25

3 authors, including:

Erik Brunvand

University of Utah

117 PUBLICATIONS **1,342** CITATIONS

SEE PROFILE

All content following this page was uploaded by Erik Brunvand on 07 October 2014.

The user has requested enhancement of the downloaded file.

IMPLICIT METHODS FOR TIMED CIRCUIT SYNTHESIS

by

Robert Thacker

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

June 1998

Copyright © Robert Thacker 1998

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Robert Thacker

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

________________ ________________________________

Chair: Chris J. Myers

________________ ________________________________

Ganesh C. Gopalakrishnan

________________ ________________________________

Erik Brunvand

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Robert Thacker in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

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 thesis 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.

To Eeyore, for his constant support and companionship.

CONTENTS

ABSTRACT iv

LIST OF FIGURES vii

ACKNOWLEDGEMENTS ix

CHAPTERS

1. INTRODUCTION 1

1.1 Related work 3
1.2 Contributions 4
1.3 Outline 5

2. STATE SPACE EXPLORATION 6

2.1 Motivating example 7
2.2 Explicit timed state space exploration 7
2.3 Implicit timed state space exploration 13
2.4 Implicit RSG representation 20
2.4.1 Reachable state space 20
2.4.2 NextState function 21
2.4.3 Existing Graphs 25
2.5 Results 27

3. SYNTHESIS 34

3.1 Excitation regions and quiescent states 34
3.2 Timed circuit implementation 37
3.2.1 Single cube covers 37
3.2.2 Multicube covers 38
3.3 Correct cover formulation 38
3.3.1 gC cover violations 39
3.3.2 SC cover violations 39
3.3.3 Correct covers 40
3.4 Results 41

4. CONCLUSIONS AND FUTURE WORK 48

REFERENCES 50

LIST OF FIGURES

1.1 ATACS design flow. 4

2.1 SPDOR (a)block diagram and (b)waveform. 7

2.2 CHP description of the SPDOR gate. 8

2.3 Self-precharging dynamic OR gate: timed ER structure. 10

2.4 RSG for the self-precharging dynamic OR gate. 11

2.5 A sample (a)Geometric region and (b) the corresponding constraint matrix. 14

2.6 Function to extract a BDD for the rule set R_m. 15

2.7 MTBDD representation of (a) R_m and (b) the number "2". 16

2.8 Function to create a MTBDD for the matrix M. 18

2.9 MTBDD representation of a geometric region matrix. 18

2.10 MTBDD representation of a timed state. 19

2.11 A false-terminated array holding (a)one, (b)two, or (c)three matrices. 22

2.12 Function to extract a BDD for the state s. 22

2.13 Self-precharging dynamic OR gate: BDD for S. 22

2.14 SPDOR graphs:(a)incorrect enablings, (b)correct enablings. 23

2.15 SPDOR circuits: (a)using incorrect enablings, (b)using correct enablings. 23

2.16 Simple diamond: (a)speed independent, (b)timed with incorrect enablings, (c)timed with correct enablings, and a transition to a "ghost state". 24

2.17 Simple diamond:(a)original TERS fragment and (b)incorrect TERS fragment. 24

2.18 Algorithm to find reachable state space S. 26

2.19 Algorithm to find NextState relation N. 26

2.20 Algorithm to add transitions to ghost states. 26

2.21 A FIFO composed of lapb elements. 28

2.22 Timed ER structure for a lazy-active, passive buffer. 28

2.23 Max memory usage for (a)*lapb4* and (b)*lapb5*. 31

2.24 BDD node usage (a) for state space and (b) overall. 33
3.1 Algorithm to find excitation regions. 36
3.2 SPDOR graphs:(a)incorrect enablings, (b)correct enablings. 36
3.3 Circuit types:(a) a standard C-element design, (b) a generalized C-element design with weak-feedback, and (c) a fully-static gC design. . . 37
3.4 Fragment of an RSG for a standard OR gate. 39
3.5 Valid cover for $c \uparrow$ for a standard OR gate. 42
3.6 Valid cover for SPDOR feedback gate. 43
3.7 Valid circuit for SPDOR feedback gate. 44
3.8 Function to synthesize circuit from a reduced state graph G 47

ACKNOWLEDGEMENTS

I would like to thank Dr. Chris J. Myers, my advisor, for the support and direction he has given during the course of this project and for tolerating my idiosyncrasies. I would also like to thank Dr. Ganesh C. Gopalakrishnan and Dr. Erik Brunvand for serving on my supervisory committee and for their invaluable assistance and comments on this research.

I am especially grateful to Dr. Steve Burns from Intel Corporation for inspiring this line of research and for many insightful discussions. The timing analysis work would have been impossible without the collaboration of Wendy Belluomini. I would also like to thank Luli Josephson, Hans Jacobson, Hao Zheng, Brandon Bachman, Eric Mercer, and Chris Krieger (my colleagues from the University of Utah) for their illuminating comments on this research. I would like to extend my thanks to Dr. David Long of AT&T Bell Labs for writing a great BDD package and to Kurt Partridge for the use of his BDDTCL package.

My parents deserve a special mention for allowing me to haunt their basement for the many years I have been in school.

This research has been brought to you by a grant from Intel Corporation, NSF CAREER award MIP-9625014, and SRC contract # 97-DJ-487.

CHAPTER 1

INTRODUCTION

千里之行始于足下

A journey of a thousand miles must begin with a single step.
- Lao-tzu, The Way of Lao-tzu

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. In fact, many modern chips have a number of communicating clock domains, which eliminate many of the advantages of a synchronous design and greatly increase design complexity. Furthermore, in most cases these designs are created in an ad hoc fashion, with little tool support for synchronization issues, and are difficult to verify. 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. Power dissipation is also reduced because gates only switch when they are doing meaningful work, instead of at every clock edge. 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 worse than the

synchronous worst-case.

Timed circuits are a class of asynchronous circuits which use explicit timing information in circuit synthesis. Although precise timing relationships are often unknown before synthesis and technology mapping, the designer usually knows some reasonable estimates. 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 [29].

Timed circuit synthesis consists of two phases. The first stage involves the exploration of the timed state space to determine which untimed states are reachable by the system. Because these state spaces grow exponentially with the number of signals, it is important to find efficient methods to store the information compiled. One method of accomplishing this is to implicitly represent the data points representing the state space.

The second stage 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 all possible circuit implementations which conform to these behaviors.

We have adapted the `ATACS` tool to use implicit methods to improve memory performance. *Binary Decision Diagrams* (BDDs) [8] are used throughout circuit synthesis. Where appropriate, *Multi-terminal Binary Decision Diagrams* [16] (MTBDDs, also known as *Algebraic Decision Diagrams* or ADDs [34]) are used to store integer valued data. BDDs are a highly effective way to store and manipulate boolean functions. MTBDDs allow this methodology to be extended to integer valued functions with boolean inputs. We have found the (MT)BDD representation to be much smaller than an equivalent explicit representation.

1.1 Related work

Many systems exist for the synthesis of untimed asynchronous circuits [21]. One methodology is the use of fundamental mode designs [37], where signals are constrained to change one at a time, and must give the system time to settle before other signals may change. Another possibility is burst-mode circuits [17, 31, 40], where this limitation is extended to allow a set (or burst) of inputs to arrive concurrently, followed by a burst of outputs. Delay-insensitive circuits are a third method, with the assumption that the delays of both wires and gates are unbounded [7, 20, 27]. Speed-independent circuits are similar, but assume that wire delays are negligible [2, 12, 26]. These synthesis methods use little or no timing information, and therefore can lead to inefficient circuits because they need to correctly handle cases which never occur in practice.

Many models have been proposed for the analysis of timed systems. These range from continuous timers on individual events to large equivalence classes representing groups of events. The timed circuit synthesis method used in `ATACS` [29] allows a lower and an upper bound to be assigned to the causal relationships between signals. Timing analysis is performed using geometric regions, which have been shown to be an efficient method for representing information about timed state spaces [4]. Unfortunately, large state spaces are still generated when the method is applied to large, complex designs, and memory size can be prohibitive.

BDDs have been shown to be an efficient way to represent design information and large state spaces. In [11], BDD techniques are developed to decompose generalized C-element circuits in a hazard free manner, the result being a parameterized description of all hazard free decompositions. In [6], implicit methods are applied to the analysis of timed systems. BDDs are used to perform discrete time analysis, with timing values represented as binary vectors. The system is unable to analyze large models due to the complexity of the discrete time model used. The COSPAN tool also uses implicit methods to perform state space analysis, but uses the unit-cube algorithm [1]. This method also suffers from state space explosion when used on relatively small designs.

1.2 Contributions

Figure 1.1 shows the design flow of the ATACS tool. The focus of this work has been to apply implicit methods to timed state space exploration and the synthesis of timed circuits within this tool framework.

MTBDDs have been used as an implicit data structure to store information compiled during state space exploration. A standard explicit state space exploration method is used, but the list of geometric regions encountered as well as the resulting state graph are represented using MTBDDs. This mixed approach of implicit structures and explicit algorithms results in a tradeoff. For large examples the memory performance of the MTBDD representation can be two to three times better than the explicit representation, but unfortunately also takes an order of magnitude more time to manipulate. In those cases where the data set would not

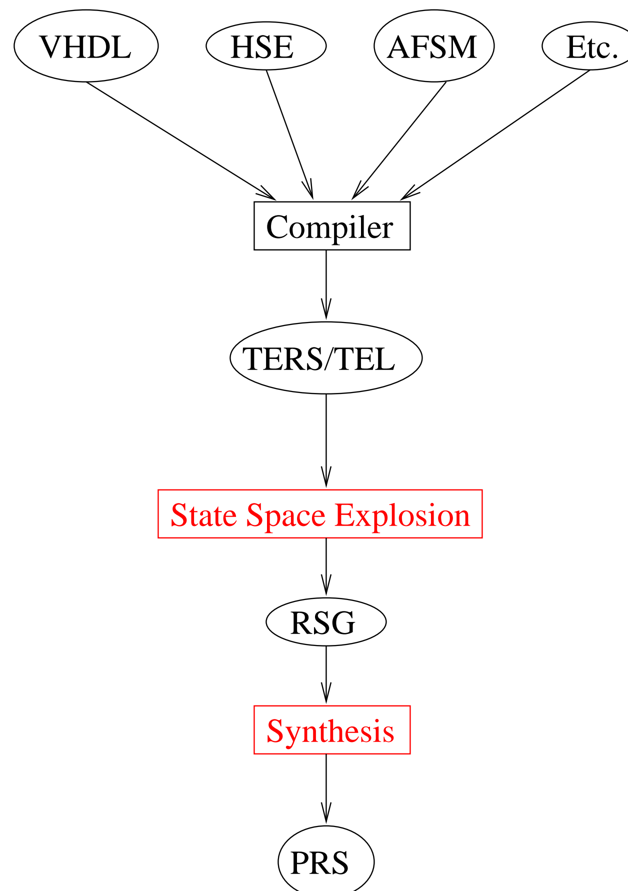


Figure 1.1. ATACS design flow.

otherwise fit in the memory of the computer, it is a win.

In the synthesis stage the entire process has been reengineered to use implicit algorithms. Runtimes for the synthesis stage using this method compare reasonably well with heuristic single-cube approaches. It also results in a substantial improvement compared with the best exact method. (Of course, this gain is dwarfed by the amount of time spent in state space exploration.) The main advantage of this approach is that it allows the derivation of solution spaces containing all valid solutions to the synthesis problem.

There is another gain which is somewhat difficult to quantify: the use of implicit methods provides a certain elegance to the coding process. The algorithms used in this work can be expressed as mathematical operations on boolean functions, and the use of a BDD package allows direct mapping to primitive operations on BDD structures. Such code is easier to develop and decode, more aesthetically pleasing, and simpler to verify.

1.3 Outline

Chapter 2 discusses issues relating to the exploration of timed state spaces. Chapter 3 discusses methodologies for timed circuit synthesis. Finally, Chapter 4 gives some conclusions and some ideas for future work.

CHAPTER 2

STATE SPACE EXPLORATION

I could be bounded in a nutshell, and count myself the king of infinite space...
- Hamlet, Act 2, Scene 2

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. In fact, in a naive algorithm where a continuous timer is associated with each signal transition, the timed state space is infinite. A slightly better representation would be to attach a clock to each signal transition that advances only in discrete time steps [10]. This does make the state space finite, but it still explodes [36]. 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 [4, 30, 35, 36]. 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 [9]. Since state space exploration is such a memory intensive process, it is an excellent candidate

for such an approach.

2.1 Motivating example

The circuit shown in Figure 2.1 is a self-precharging dynamic OR gate (SPDOR) and is used as an example throughout this chapter. Figure 2.1(a) shows a block diagram of the circuit, and Figure 2.1(b) shows the waveforms which describe the behavior of the circuit. Briefly, the circuit receives a pulse from either i_1 or i_2, and reacts with a pulse on the output a. A rising transition on the output causes the feedback signal x to fall, which causes the reset of signal a. The falling transition on a then sets x high, and the gate is ready to process another pulse. The timing annotated handshaking expansion for this circuit is shown in Figure 2.2.

2.2 Explicit timed state space exploration

The state space exploration procedure used by `ATACS` begins with a *timed event-rule (ER) structure*, described formally in [4, 29]. Timed ER 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 timed ER 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 behavior. Rules are annotated with a bounded timing constraint which must be satisfied in order to enable a transition to occur. Each rule is of the form $\langle e, f, l, u \rangle$, where e is the *enabling event*, f is the *enabled event*, and $\langle l, u \rangle$ is the

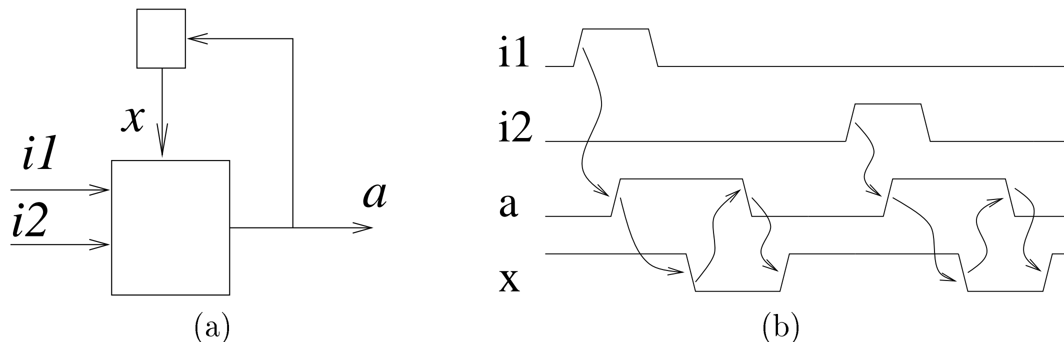


Figure 2.1. SPDOR (a)block diagram and (b)waveform.

```
module synor;

delay idelay = <500,550;269,299>;
delay xdelay = <101,111;99,109>;
delay adelay = <201,221;199,229>;

input i1 = {false,idelay};
input i2 = {false,idelay};
output a = {false,adelay};
output x = {true,xdelay};

process a;
*[[ i1+ -> a+; x-; a-; x+
  | i2+ -> a+; x-; a-; x+
 ]]
endprocess

process ienv;
*[[ skip -> i1+; i1-
  | skip -> i2+; i2-
 ]]
endprocess

endmodule
```

Figure 2.2. CHP description of the SPDOR gate.

bounded timing constraint. 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 timed ER structure for the SPDOR gate is shown in Figure 2.3. Nodes represent signal transitions and arcs represent causal relationships between them. Each arc should be annotated with a set of timing bounds, but in this drawing most have been removed to keep things simple. Tokens on arcs indicate that the preceding transition has occurred but the following transition has not. Except where signals are in conflict, all incoming arcs must have tokens to fire a transition. Where two signals conflict but both have causal arcs to the same event, only one of the two tokens need be present to cause the transition. In a similar fashion, when a signal transition causes two conflicting events, tokens are placed on both arcs but the occurrence of one of the conflicting events removes the token enabling the other.

The goal of state space exploration is to derive the *state 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 $s : I \cup O \to \{0, R, 1, F\}$

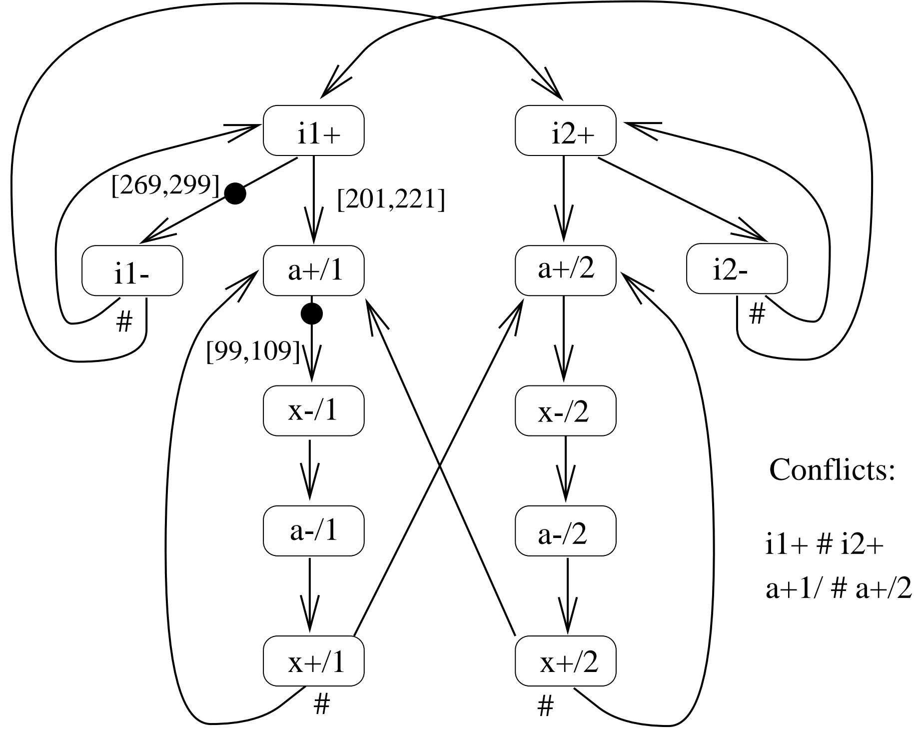


Figure 2.3. Self-precharging dynamic OR gate: timed ER structure.

which returns the value of each signal and whether it is enabled, i.e.,

$$s(x) \equiv \begin{cases} 0 & \text{if } x \text{ is stable low in } s \\ R & \text{if } x \text{ is enabled to rise in } s \\ 1 & \text{if } x \text{ is stable high in } s \\ F & \text{if } x \text{ is enabled to fall in } s \end{cases}$$

It is useful to also define a function *val* which strips the excitation information, i.e.,

$$val(s(x)) \equiv \begin{cases} 0 & \text{if } s(x) = 0 \text{ or } s(x) = R \\ 1 & \text{if } s(x) = 1 \text{ or } s(x) = F \end{cases}$$

Finally, the predicate *enabled* returns true if the signal is enabled, i.e.,

$$en(x) \equiv (x = R \ \vee \ x = F).$$

Traditional definitions of state labeling functions have not included the enabling of signals as it can usually be inferred from the set of state transitions. In timed circuits, however, it is possible that a signal is enabled, but another signal always fires first. In this case, there would be no state transition out of that state in which that signal fired, and thus, it would not be possible to infer from the state graph

that the signal is enabled. This information is necessary to properly synthesize timed circuits for the output signals. The notation 1* (or 0*) has also been used to indicate that a signal is enabled to change.

A state graph is defined to be well-formed if for any state transition (s, s') in Γ, the value of exactly one (denoted by $\exists!$) enabled signal in s changes to a new value in s'. , i.e.,

$$(s, s') \in \Gamma \Rightarrow \exists! x \in I \cup O . val(s(x)) \neq val(s'(x))$$

The signal x that differs in value in the state transition (s, s') is denoted as follows: $s \xrightarrow{x} s'$. Our synthesis procedure also requires that the state graph be *complete state coded*, defined to be that for any two states in which all signals have the same value, any output signal enabled in one state is also enabled in the other [12].

Figure 2.4 shows the RSG describing the behavior of the SPDOR circuit. In the initial state (RR10) both $i1$ and $i2$ are enabled to rise, while a is stable low and x is stable high. This state may be exited either on the transition $i1 \uparrow$ or $i2 \uparrow$. Note that the transitions $i1 \uparrow$ and $i2 \uparrow$ are in conflict, as indicated in Figure 2.3; one or the other may occur, but not both. The occurrence of $i1 \uparrow$ therefore disables $i2 \uparrow$, and we enter state (F01R), not (FR1R). In this state, either a may rise or $i1$ may

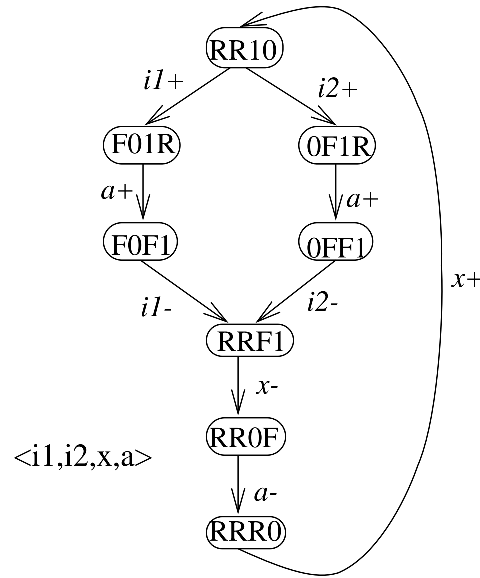


Figure 2.4. RSG for the self-precharging dynamic OR gate.

fall, while $i2$ is stable low, and x is stable high. Notice that there is no edge for the transition $i1 \downarrow$. As shown in Figure 2.3, the maximum delay for a rising is 221 time units, while the *minimum* delay for $i1$ falling is 269 time units. This determines that $a \uparrow$ always occurs first, so (001R) is eliminated as a reachable state. Firing this transition causes the system to enter state (F0F1). This state is represented by the enabling tokens shown in Figure 2.3. Again, although sufficient tokens are present to fire both the event i1- and the event x-/1, the timing indicates that the maximum time from i1+ to i1- is 299 time units, while x- occurs no less than 300 time units after i1-. Therefore, i1- always occurs first and the system enters state (RRF1).

While the state space of this system is relatively easy to find, the time constraints on most systems are more difficult to analyze. It is important to have an efficient algorithm to perform this analysis. The method used is to perform a depth first search to find all reachable timed states. A timed state for an ER 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 vector s_c defines an *untimed state* and contains a variable for each signal in the system. These variables may take on any one of the following values: 0 denotes a stable low signal, R denotes a signal enabled to rise, 1 denotes a stable high signal, and F denotes a signal enabled to fall. The timing information, TI, is represented with geometric regions, which were first introduced in [5, 19, 24].

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. This constraint matrix represents a convex n-dimensional region, where n is the number of enabled rules. Each dimension corresponds to a rule, and the firing times of the enabled events for the rules can be anywhere within the space. Figure 2.5(a) shows a sample geometric region, and Figure 2.5(b) shows the corresponding constraint matrix. Again, 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 2.5(a) contains all points (t_2, t_1) which conform to these timing constraints.

In order to track progress through the timed state space, it is necessary to record the geometric regions encountered for each state so that previously explored paths are not repeated. The explicit method maintains the timed state list and the state graph in a joint structure, a hash table where each entry is an augmented timed state (a timed state with transition links). Each entry in the state table contains the s_c vector and the R_m set for that state with a linked list of associated geometric regions. Pointers are stored with each state to indicate its predecessor and successor states.

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

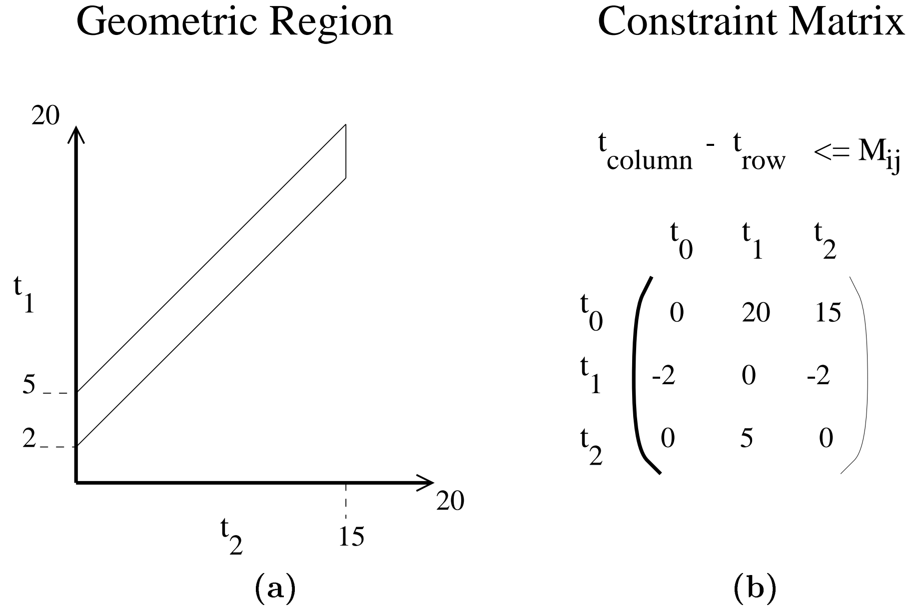


Figure 2.5. A sample (a)Geometric region and (b) the corresponding constraint matrix.

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 [8]. Because geometric region information is integer-valued, MTBDDs have been chosen to store the region matrices. MTBDDs are a type of BDDs 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 have been 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 first step in building the representation is to use BDDs to store the bit vector that indicates which rules are in R_m. To accomplish this, an atomic BDD is allocated to represent each rule. These BDDs are assembled into the array

$\mathbf{m} = (m_1, ...m_n)$, where n is the number of rules in the timed ER structure. An atomic BDD is one which represents a single variable. As shown in Algorithm 2.3.1 (see Figure 2.6), a new BDD, β, is created with the value TRUE. Each member of the rule set R is then considered. If that rule is a member of the R_m set, the corresponding m_i BDD is added to β, otherwise the complement of the appropriate m_i BDD is added. The resulting BDD uniquely represents the R_m set. In an ER structure with four rules, where $R = \{r_1, r_2, r_3, r_4\}$, $\mathbf{m} = (m_1, m_2, m_3, m_4)$, and $R_m = \{r_1, r_3\}$, (meaning that rules 1 and 3 are enabled, but rules 2 and 4 are not), the implicit representation of the set of enabled rules would be composed of the product $m_1 \wedge \overline{m_2} \wedge m_3 \wedge \overline{m_4}$ and is shown in Figure 2.7(a). (Note that BDDs as shown in this thesis are drawn to be relatively readable, and do not necessarily indicate the actual node ordering or machine representation of these structures.)

It is also 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 ith matrix associated with a given R_m set. Any number i can be viewed as a bit vector $\vec{i} = (i_0, ..., i_n)$, where i_0 is the low order bit of the binary representation of i, and i_n is the high order bit. A set of BDD variables is used to represent the binary value of i, and a number BDD is constructed in a manner analogous to that used for the R_m set. For instance, the BDD shown in Figure 2.7(b) represents the number "2" in a four-bit notation. Numbers of this form are used to

Algorithm 2.3.1 (Extract R_m BDD)
bdd FindR$_m$BDD(rule set R, rule set R$_m$, bdd array $\mathbf{m}$*)* {
  bdd $\beta = TRUE$
  foreach *($r_i \in R$)*
    if *($r_i \in R_m$)* **then**
      $\beta = \beta \wedge \mathbf{m}[i]$
    else
      $\beta = \beta \wedge \neg\mathbf{m}[i]$
  return β
}

Figure 2.6. Function to extract a BDD for the rule set R_m.

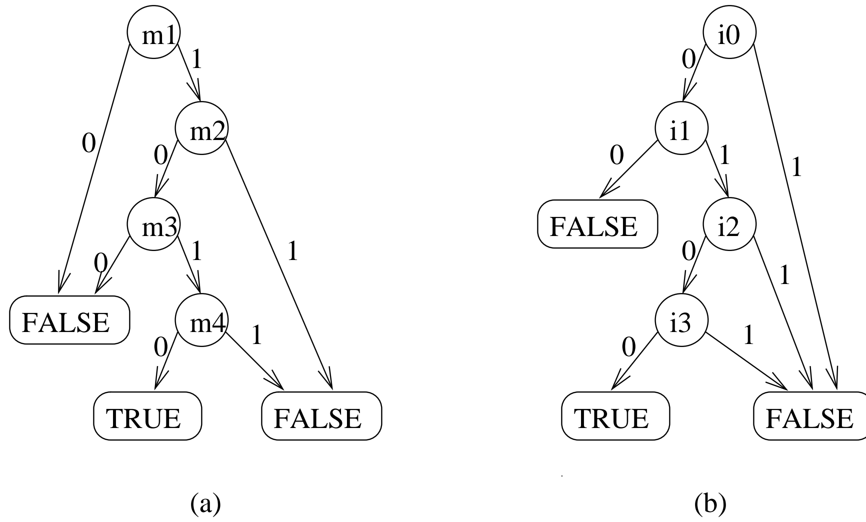


Figure 2.7. MTBDD representation of (a) R_m and (b) the number "2".

create a dynamically sized array of matrices. In order to conserve space, precisely enough bits are used to represent the largest number currently needed.

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. Similarly the representation of numbers with BDD bit vectors is extremely inefficient for small numbers of integers, but the costs are spread if many are used, and the use of this format is necessary to enable this matrix representation.

A matrix with integer entries can be viewed as a function ($N \times N \mapsto Z$), which takes row and column indices and returns the appropriate matrix entry($M(r, c) =$

M_{rc}). A square matrix can also be viewed as a function from boolean values to integers, $\{0,1\}^n \times \{0,1\}^n \mapsto Z$. The row and column indices of the geometric region matrices are thus parameterized. Each is represented as a boolean vector $\vec{r} = (r_0, r_1, r_2, ..., r_n)$ or $\vec{c} = (c_0, c_1, c_2, ...c_n)$, so the function can be viewed as $M(\vec{r}, \vec{c}) = M_{rc}$. MTBDDs are an ideal way to represent this type of function [15]. BDDs are constructed for each necessary row and column index, and stored in arrays **r** and **c**. The BDD for the ith column index is stored in **c**[i] and the BDD for the i^{th}row index is stored in **r**[i]. For example, **r**[3] represents the value "3" using a set of variables which indicate that it is a row index. Each augmented matrix is then transformed into a MTBDD. Figure 2.8 shows the algorithm used to accomplish the transformation. First, β is initialized to FALSE. Then each matrix location is considered in turn. If that location is not tagged as "not an entry", the BDD α is set to represent the appropriate indices and a terminal node is created with the proper value. The entry is then inserted into the matrix BDD using the ITE operator. This operator takes three parameters: the first must be a normal BDD, and the others may be either MTBDDs or normal BDDs. The effect of the call $ITE(\alpha, \gamma, \beta)$ is to take all paths in α which lead to TRUE and link them to γ, and all paths in α that lead to FALSE and link them to β. (This is equivalent to the operation $(\alpha \wedge \gamma) \vee (\neg\alpha \wedge \beta)$ if all parameters are normal BDDs.) Since any path not leading to a valid terminal ends in FALSE, there is no need to explicitly link "not an entry" locations. Figure 2.9 shows the MTBDD representation of the following matrix:

$$\begin{pmatrix} 0 & 20 & x & 15 & x \\ -2 & 0 & x & -2 & x \\ x & x & x & x & x \\ 0 & 5 & x & 0 & x \\ x & x & x & x & x \end{pmatrix}$$

Since rows 2 and 4 and columns 2 and 4 are filled with "not an entry" (and since there is no row or column 5, 6, or 7), the BDD representation truncates those paths with FALSE as soon as possible. Matrices represented in this form can be compared for equality by checking to see if they are the same MTBDD, which is a simple pointer check.

Algorithm 2.3.2 (Construct Matrix MTBDD)

mtbdd MakeMatrixBDD(int n, matrix M, bdd vector $\mathbf{r}$*, bdd vector* $\mathbf{c}$*)* {
  mtbdd $\beta = FALSE$
  forall $(i : 0 \leq i \leq n)$
    forall $(j : 0 \leq j \leq n)$
    {
      if $M[i, j] \neq$ "*not_an_entry*" **then**
      {
        bdd $\alpha = \mathbf{r}[i] \wedge \mathbf{c}[j]$
        mtbdd $\gamma = terminal(M[i, j])$
        $\beta = ITE(\alpha, \gamma, \beta)$
      }
    }
  return β
}

Figure 2.8. Function to create a MTBDD for the matrix M.

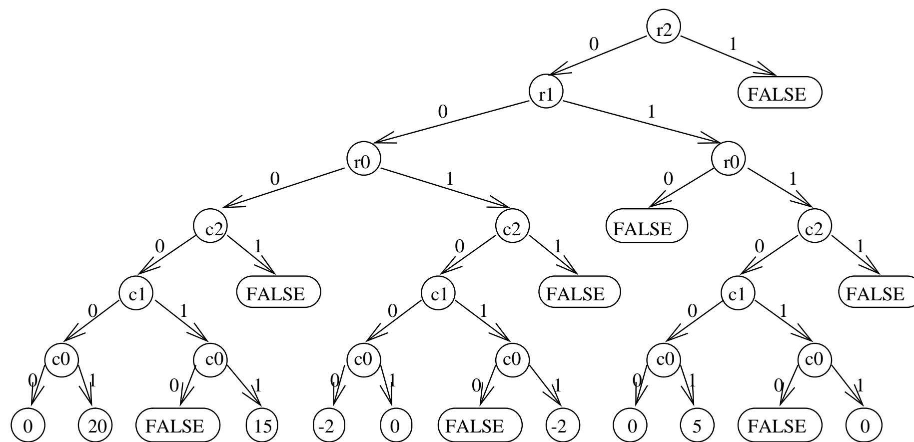


Figure 2.9. MTBDD representation of a geometric region matrix.

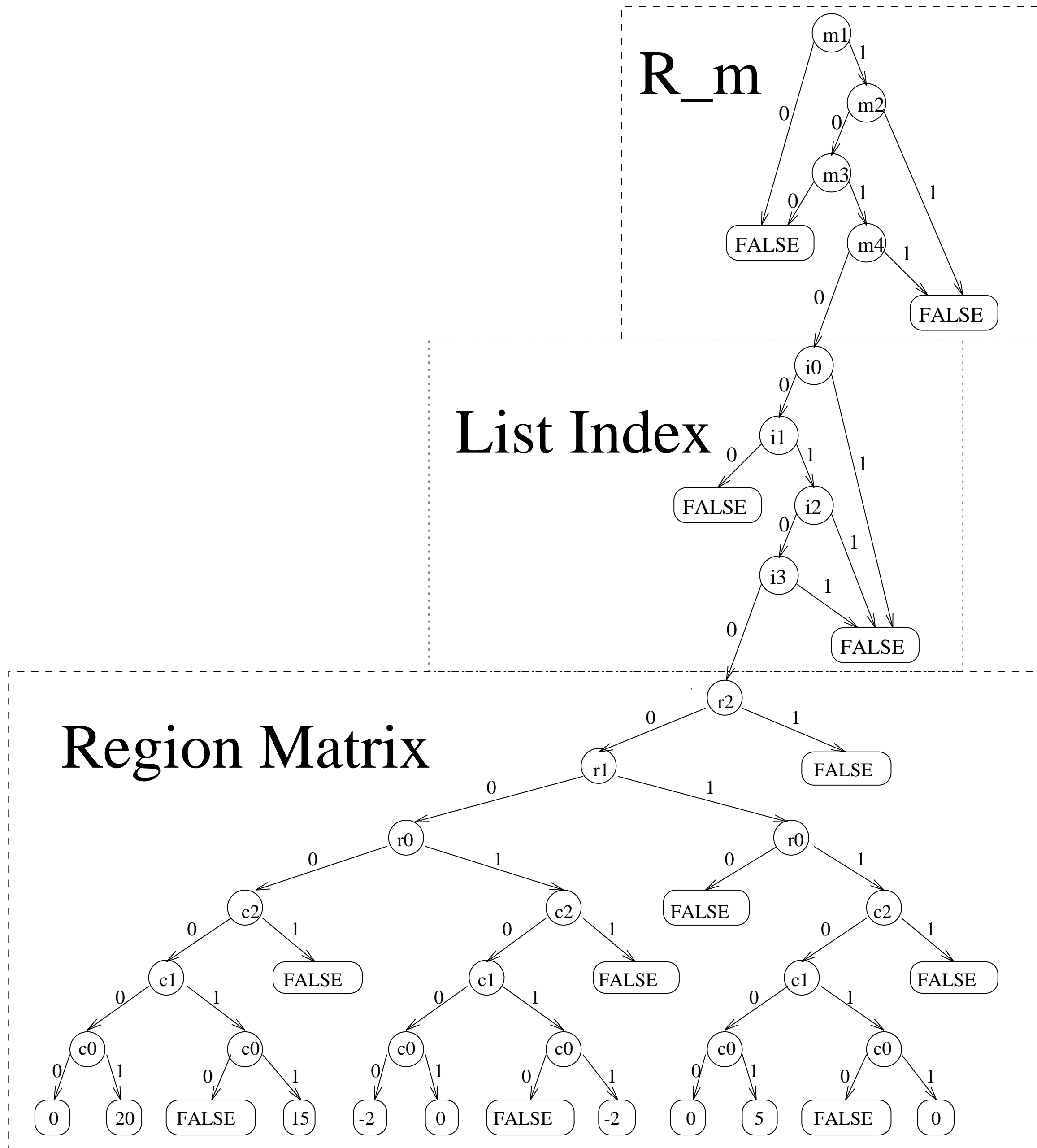


Figure 2.10. MTBDD representation of a timed state.

A timed state is represented by a composition of BDDs, one for the R_m set, another for the list index, and a third representing the geometric region matrix. Figure 2.10 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 the above matrix. When a new timed state is found, the timed state list MTBDD T_S is extended by the call

$$T_S = ITE(FindR_mBDD(R, R_m, \mathbf{m}) \wedge i, MakeMatrixBDD(n, M, \mathbf{r}, \mathbf{c}), T_S),$$

where i is the list index BDD for this region. Since list indices are kept as small as possible, a size check is made before adding this region to the array. If necessary, an extra bit (leading zero) is added to existing entries to accommodate the new growth. As shown in Figure 2.11, the index numbers are dynamically grown as the list lengthens. Index bits which do not appear in the figure are don't cares, so matrix "zero" as shown in Figure 2.11(a) also appears as every even numbered matrix. Since the list is always traversed in order, the array is FALSE terminated (much like a C string) so that the end of the array can be detected by the algorithm. When inserting matrix "one", the existing structure is first restricted to require a two bit "zero" and then matrix one is ORed in, resulting in the structure shown in Figure 2.11(b). Note that adding a third matrix (as shown in Figure 2.11(c)) does not require the use of an additional bit, but adding a fourth matrix would result in a five element list, (including the terminator), requiring three bits.

2.4 Implicit RSG representation

To construct an implicit representation of the RSG, each state is inserted as it is encountered in the exploration.

2.4.1 Reachable state space

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 $I \cup O$. S is represented using a BDD which is constructed by defining a variable for each signal, and then sweeping through the states defined in the RSG. In each state, we AND

the extracted values of the signals together to define the state (see Algorithm 2.4.1 in Figure 2.12), then OR the individual states together to create S:

$$S = S \vee FindStateBDD(s_i, \mathbf{x}).$$

Figure 2.13 shows the BDD for the state space predicate for the SPDOR example. The BDD S shows that the reachable states are those in which (1) both i1 and i2 are low, or (2) exactly one of i1 and i2 are high and x is also high.

2.4.2 NextState function

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. N is constructed in a manner analogous to S. A product term is created for each valid pair of states (s, s'), and these are ORed together to form N:

$$N = N \vee (FindStateBDD(s, \mathbf{x}) \wedge FindStateBDD(s', \mathbf{x}'))$$

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 SPDOR example, a naive derivation of S and N actually represents the state graph found in Figure 2.14(a). This graph correctly describes the signal changes, but not the enablings. A correct graph is shown in Figure 2.14(b). To illustrate the difference, the circuits synthesized from these graphs are shown in Figure 2.15. The naive derivation results in the circuit found in Figure 2.15(a). This circuit may work, but is larger and slower than the circuit shown in Figure 2.15(b), 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) stored as characters. However, in the implicit method the use of bit vectors makes this less attractive, as it would double the necessary length of the vectors.

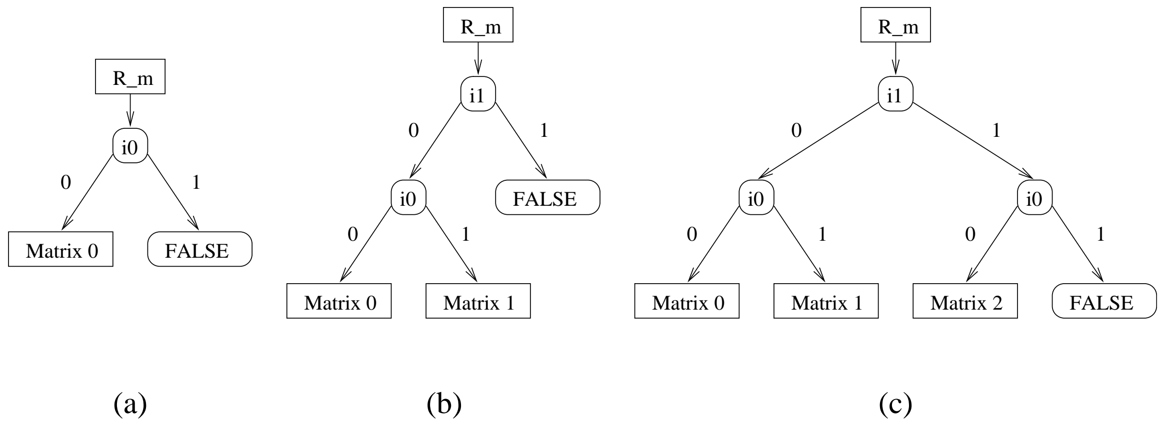


Figure 2.11. A false-terminated array holding (a)one, (b)two, or (c)three matrices.

Algorithm 2.4.1 (Find State BDD)
bdd FindStateBDD(state s, signal vector **x***)* {
  $\beta = TRUE$
  Foreach x_j *in* **x**
    if $(val(s(x_j)))$ **then**
      $\beta = \beta \wedge bdd(x_j)$
    else
      $\beta = \beta \wedge \neg bdd(x_j)$
  Return β
}

Figure 2.12. Function to extract a BDD for the state s.

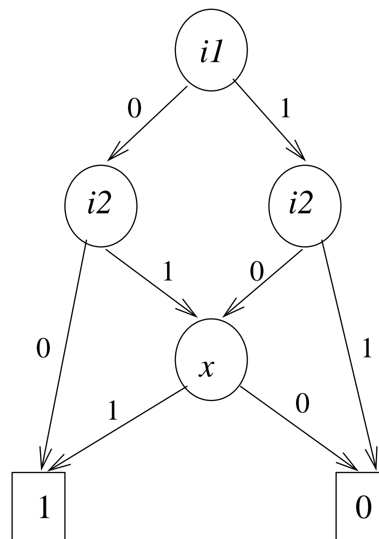


Figure 2.13. Self-precharging dynamic OR gate: BDD for S.

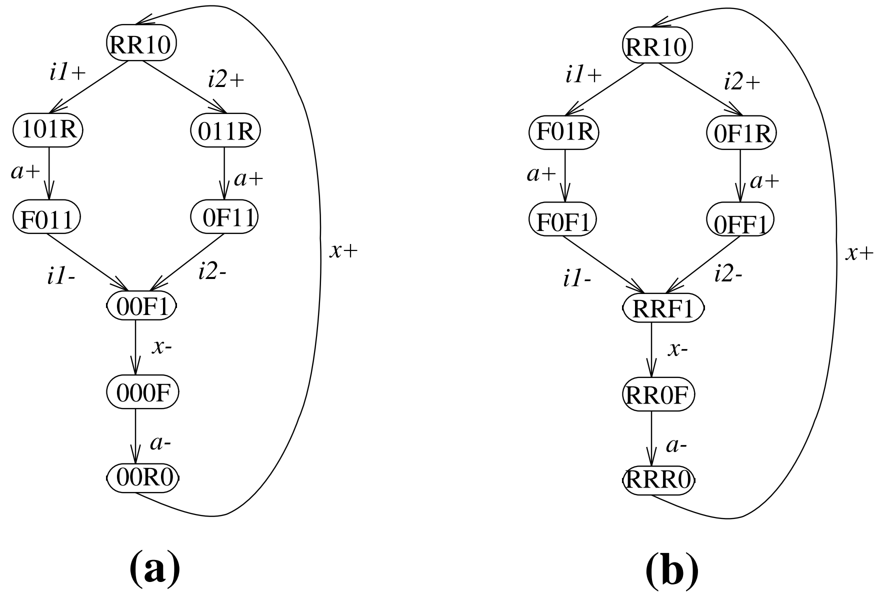


Figure 2.14. SPDOR graphs:(a)incorrect enablings, (b)correct enablings.

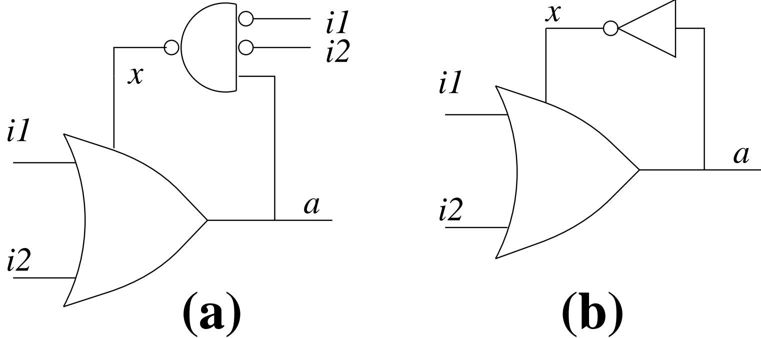


Figure 2.15. SPDOR circuits: (a)using incorrect enablings, (b)using correct enablings.

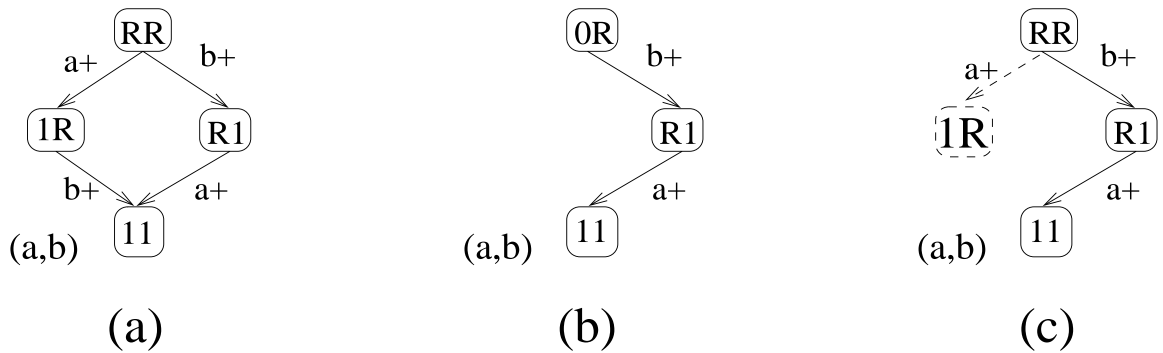


Figure 2.16. Simple diamond: (a)speed independent, (b)timed with incorrect enablings, (c)timed with correct enablings, and a transition to a “ghost state”.

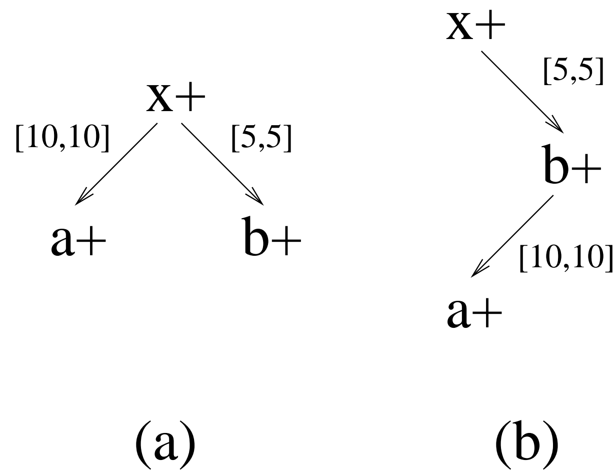


Figure 2.17. Simple diamond:(a)original TERS fragment and (b)incorrect TERS fragment.

The basic problem can be illustrated using the familiar diamond shown in Figure 2.16. The original speed-independent graph is shown in Figure 2.16(a). Because timing analysis says that the signal b always rises before a, the state (1R) is removed from the graph. If the correct enablings are not maintained, the less concurrent graph shown in Figure 2.16(b) 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. Suppose that the original graph fragment shown in Figure 2.16(a) represents the behavior shown by the TERS fragment in Figure 2.17(a): some signal x enables both a and b, with a to follow in 10 time units and b to follow in 5. The total time necessary to traverse this graph from state (RR) to state (11)

should be 10 time units. An improperly pruned graph (Figure 2.16(b)) loses the fact that x was the enabling event, and actually represents the TERS fragment in Figure 2.17(b). In this case the total traversal time has increased from 10 to 15 time units. This less concurrent circuit may not only be slower, but it may also be incorrect if it violates the original timing assumptions. Some other state may have been pruned as unreachable based on the timing of this segment. Such unreachable states are used as "don't cares" during the synthesis process. If such a state were used in minimizing a gate, and this new timing made it reachable again, there would be a hazard introduced in the system.

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 (see Algorithm 2.4.4 in Figure 2.20).

Figure 2.16(c) 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. In the SPDOR example, several ghost states are necessary, such as (001R) which has a transition from (F01R).

2.4.3 Existing Graphs

The original version of this system extracted implicit representations from existing explicit state graphs. These algorithms have been maintained to allow the system to import graphs from other systems for synthesis. The S relation is constructed using Algorithm 2.18. The N relation is constructed using Algorithm 2.19. To maintain the correct enablings, a transition to a "ghost state" is added to the RSG whenever an enabling is found without a matching next state, as shown in Algorithm 2.4.4 (see Figure 2.20). Note: we use the notation $s\mid_{x_i=1}$ to define the state where all values are the same as in s, except that the signal x_i has the value 1.

Algorithm 2.4.2 (Find Reachable State Space)
/ Given the graph G, with set of states* $s_i \in \Phi$*, find BDD S.*/*
bdd FindStateGraphBDD(RSG G) {
  S = FALSE
  Foreach s_i *in* Φ
    $S = S \vee FindStateBDD(s_i, \mathbf{x})$;
  return *(S)*
}

Figure 2.18. Algorithm to find reachable state space S.

Algorithm 2.4.3 (Find NextState Relation)
*/*Given the graph G, with set of state pairs* $(s, s') \in \Gamma$*, find BDD N.*/*
bdd NextState(RSG G) {
  N = FALSE
  Foreach (s, s') *in* Γ
    $N = N \vee (FindStateBDD(s, \mathbf{x}) \wedge FindStateBDD(s', \mathbf{x}'))$;
  return *(N)*
}

Figure 2.19. Algorithm to find NextState relation N.

Algorithm 2.4.4 (Add Ghost State Transitions)
*/*Given the graph G, with set of state pairs* $(s, s') \in \Gamma$*, add missing transitions.*/*
RSG Haunt(RSG G) {
  Foreach s_i *in* Φ
    Foreach x_j *in* $\mathbf{x}$
      if $((s_i(x_j) = R) \wedge ((s_i, s_i\,|_{x_j=1 or F}) \notin \Gamma))$
        $\Gamma = \Gamma \cup \{(s_i, s_i\,|_{x_j=1})\}$;
      else if $((s_i(x_j) = F) \wedge ((s_i, s_i\,|_{x_j=0 or R}) \notin \Gamma))$
        $\Gamma = \Gamma \cup \{(s_i, s_i\,|_{x_j=0})\}$;
  return *(G)*
}

Figure 2.20. Algorithm to add transitions to ghost states.

2.5 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 also parameterized two asynchronous FIFO examples in order to demonstrate the effectiveness of implicit methods. One, described below, is a simple lazy-active passive buffer (*lapb*). The other is a parameterized version of the high-performance FIFO element described in [28](referred to as *fifo*). The lazy-active-passive buffer FIFO is constructed of a chain of lazy-active-passive buffers which behave as FIFO elements. The buffer continually reads data from its left port and sends data to its right port, implementing the CHP $*[L?; R!]$, illustrated graphically in Figure 2.21. Figure 2.22 shows a timed ER structure that specifies a *lapb* implemented with a four-phase communication protocol. The signal li is the buffer's input on the left channel, lo is the output on the left channel, ri is the input on the right channel, and ro is the output on the right channel. A state variable a is also included in the specification to allow it to have complete state coding (CSC) [12]. A number of timing assumptions are made in the specification to optimize the circuit and are shown as ranges attached to each rule. Rules that enable transitions on li are given a delay range $[l_L, u_L]$, which indicates that this range is set depending on what the *lapb* element is communicating with. If it is communicating with another similar *lapb* circuit, this range is $[1, 5]$ like the rest of the ranges. If the circuit is communicating with a dissimilar circuit, these ranges are set to $[100, \infty]$, since the behavior of the environment is assumed to be slow. The $[l_R, u_R]$ delay ranges used on rules enabling transitions in ri are assigned in a similar way. Both this FIFO and the one described in [28], are very concurrent when parameterized and generate an extremely large number of geometric regions which correspond to the number of regions necessary to synthesize a large complex design.

Table 2.1 shows the results of applying both explicit and implicit state space representation techniques to the various examples. The partial order method for state exploration discussed in [4] is used to generate the timed state space. The

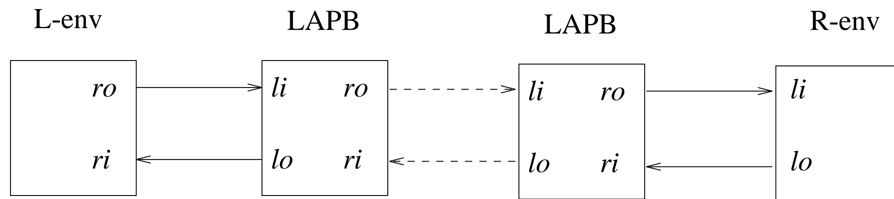


Figure 2.21. A FIFO composed of lapb elements.

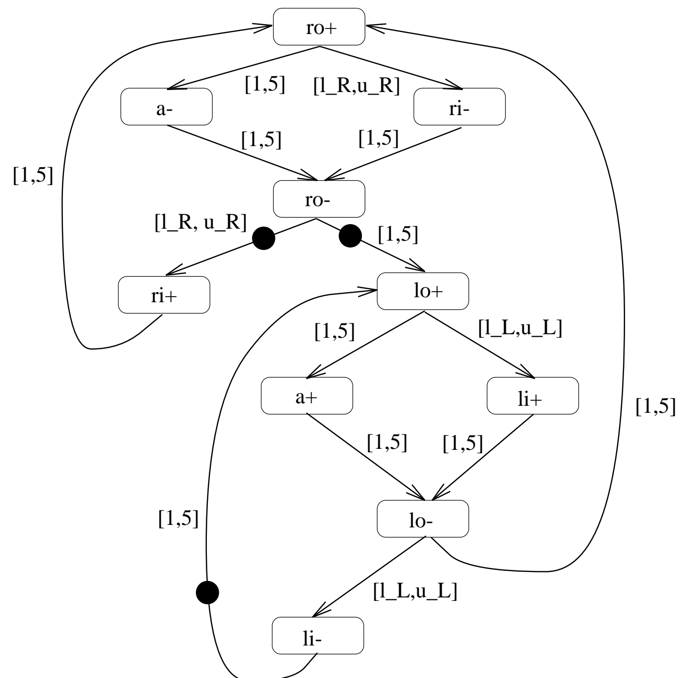


Figure 2.22. Timed ER structure for a lazy-active, passive buffer.

examples shown were run on an Sparc20 with 128Mb of physical memory. The table is divided into two sections: the top section contains information about various timed circuit benchmarks, and the bottom section contains data on the parameterized examples *lapb* and *fifo*. The first column in the table shows the number of regions that are found for each specification. From this column, it is clear that adding each new stage to the *fifo* or *lapb* examples causes the number of regions to increase exponentially. The Max Memory columns depict the maximum amount of memory in megabytes that is used during state space exploration. This number is the factor that limits the size of specifications that can be synthesized. The table shows that for small examples, such as the one and two element *lapb*s and *fifo*s, as well as examples in the first part of the table, implicit methods do not improve and sometimes even worsen this performance measure. However, for small examples,

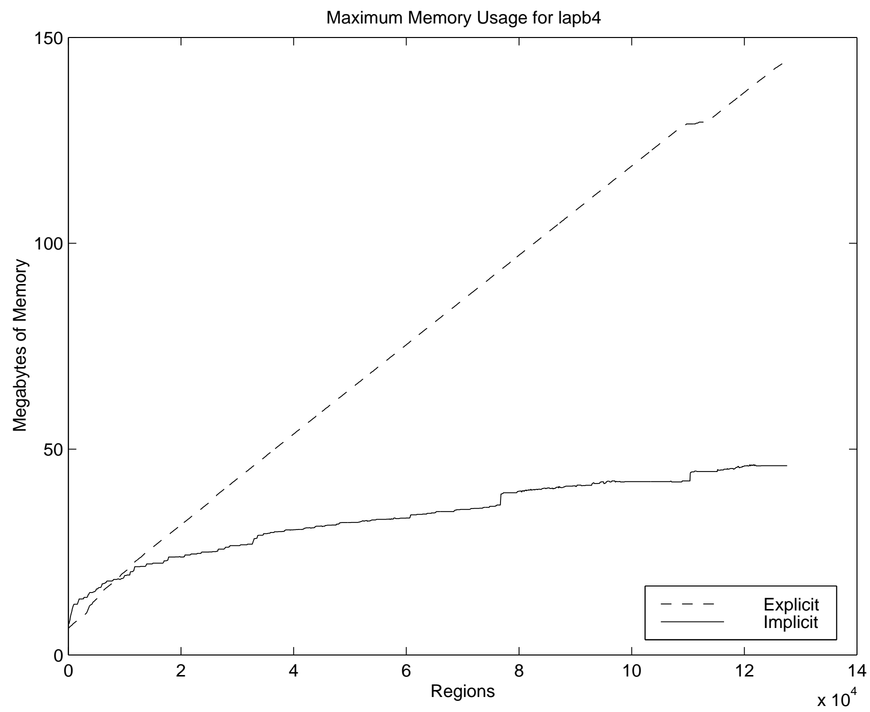
Table 2.1. Experimental results. Memory values are given in Mb.

		Implicit Rep.		Explicit Rep.	
		Mem Max	CPU	Mem Max	CPU
Examples	Regions	(Mb)	Time	(Mb)	Time
spdor	21	.89	.24	.82	.026
spdand	91	1.04	1.07	.82	.16
cnt	171	2.0	6.32	1.6	.57
mmuoptSV	955	1.8	38.8	1.7	6.6
mmuopt	149	1.3	3.47	.95	.45
slatch	68	1.2	1.14	.9	.13
elatch	115	1.3	2.28	.9	.24
SELopt	1116	2.5	62.8	2.7	7.1
tsbm	1784	3.0	74.0	2.9	11.3
scsiSVT	20	.84	.20	.77	.016
lapb	56	.96	.45	.79	.089
lapb2	615	1.7	11	1.3	1.5
lapb3	8226	5.5	500	8.0	61
lapb4	127,618	40.1	4.3×10^4	143	6.6×10^3
fifo	81	1.2	1.65	.90	.163
fifo2	828	2.5	36	1.9	4.3
fifo3	12371	12	1683	17	175

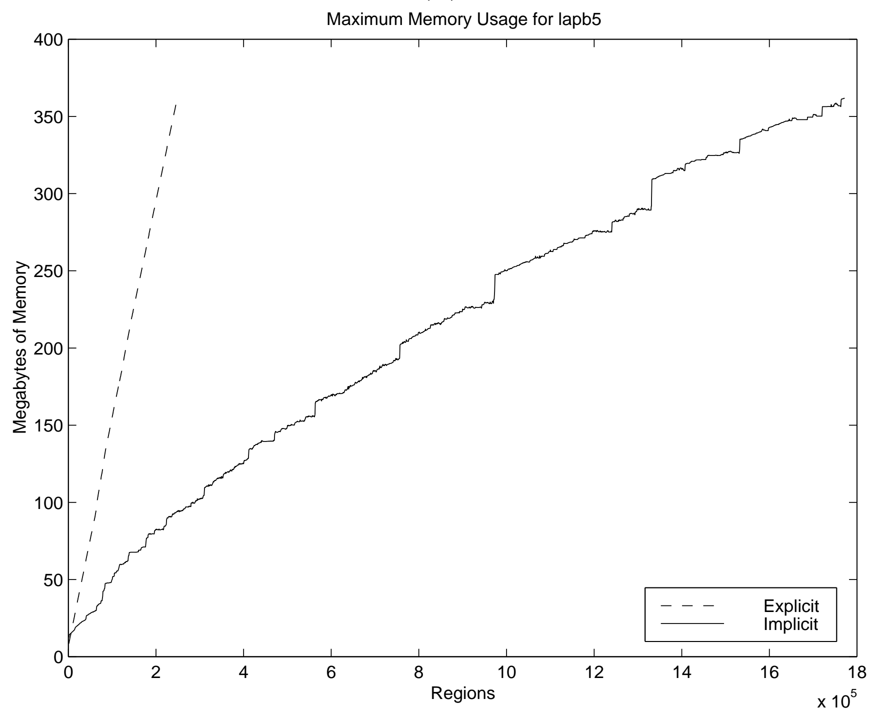
memory size is not an issue since modern machines regularly contain at least 32Mb of memory. In the larger examples, *lapb3*, *lapb4*, and *fifo3*, the benefits of implicit methods become clear. On *lapb3* and *fifo3*, the implicit representation only requires about two thirds of the memory required by the explicit representation. On *lapb4* the explicit representation requires less than a third of the memory required by the explicit representation. The columns labeled CPU time show the amount of time spent in state space exploration for each method. The implicit method normally takes approximately 10 times as long as the explicit method. On large examples, however, the implicit method takes much less space and is able to complete examples which the explicit algorithm cannot do.

Figure 2.23 shows the memory usage pattern of the state space exploration of *lapb4* and *lapb5* 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 implicit methods are much better. As the number of regions grows very large, the amount of memory used by the implicit methods 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 2.23(b) does not represent a complete exploration of *lapb5*. Both methods were allowed to progress until they had exhausted the physical memory available on a Pentium II workstation with 384MB of physical memory.)

Figures 2.24(a) and 2.24(b) show the number of BDD nodes per region that



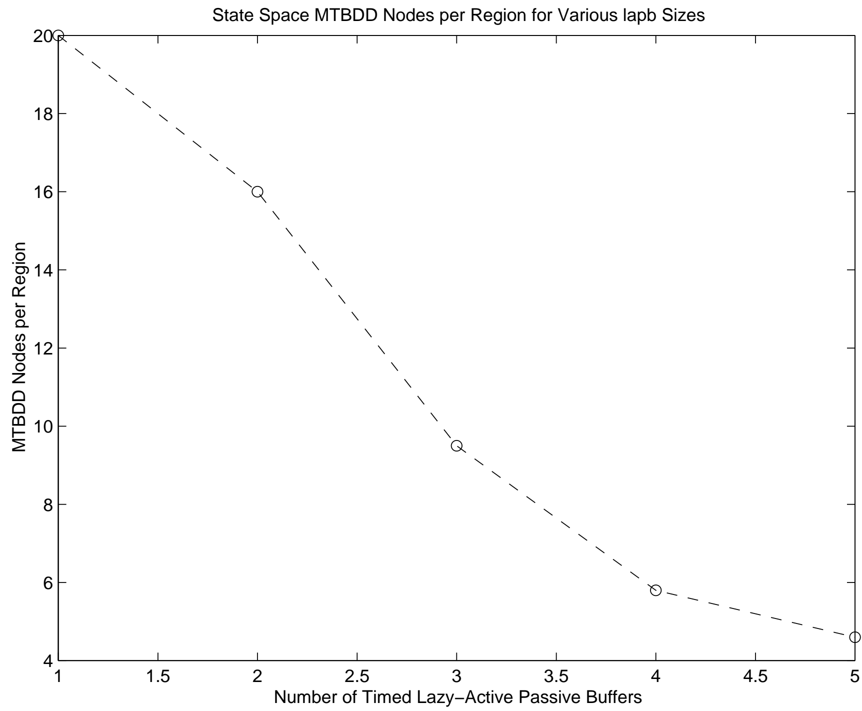
(a)



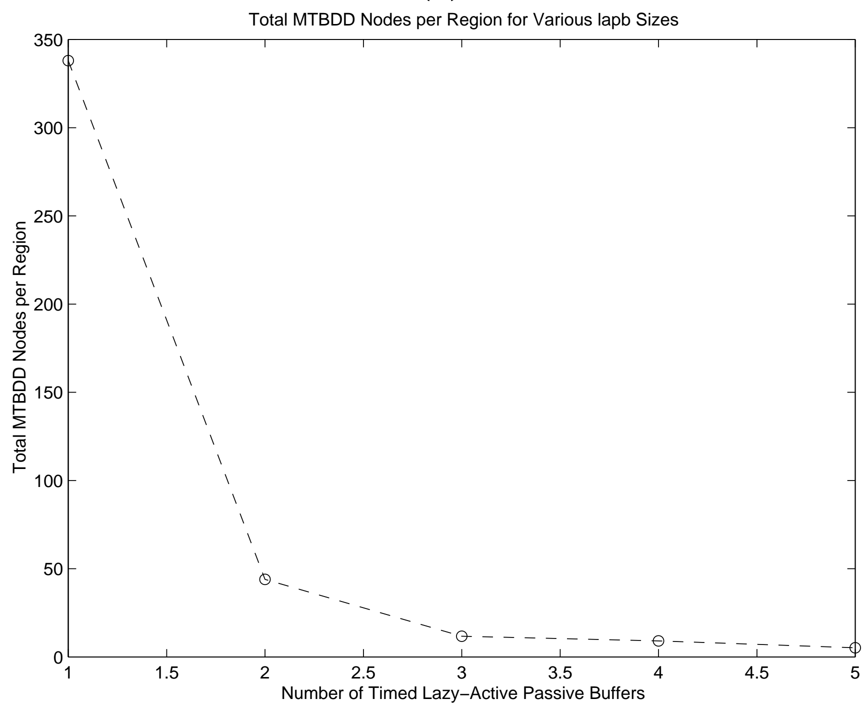
(b)

Figure 2.23. Max memory usage for (a)*lapb4* and (b)*lapb5*.

are required to do timed state space exploration for the *lapb* FIFOs of various sizes. Figure 2.24(a) shows the BDD nodes that are used in representing the timed state space, and Figure 2.24(b) shows the total BDD nodes necessary, including S, N, and overhead required to manage the BDDs. The trend on these results is also very good. As the size of the example increases, the number of nodes per region decreases dramatically, indicating that BDDs should be able to be used for even bigger examples. Unfortunately, the technique is currently limited in our implementation by the memory used in the stack. As examples get very large, a greater percentage of memory is being used storing stack elements. Most of the information on the stack can be represented using BDDs, and in the future we plan to extend this work to include that optimization. When the stack is implemented with BDDs we expect to be able to do even larger examples.



(a)



(b)

Figure 2.24. BDD node usage (a) for state space and (b) overall.

CHAPTER 3

SYNTHESIS

"Contrariwise," continued Tweedledee, "If it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."
-Lewis Carroll,
Through the Looking Glass

The synthesis stage starts with a *reduced state graph* (RSG), as described in the preceding chapter. State graphs are a common intermediate form for most asynchronous CAD tools [13, 14, 23, 26, 32, 33, 38, 39], and can be derived from many higher-level languages such as CHP and STGs [29], as well as more recently VHDL [41]. Because of this commonality, support has been included in the tool to import SGs derived from other CAD tools. `ATACS` implicitly stores RSGs using two BDD structures: S, which represents the reachable state space, and N, which describes 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 is indexed with the variable k and the kth 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*, 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:

$$QS(x\uparrow) = exist_q(\mathbf{x}', x \wedge x' \wedge N)$$

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 (see Algorithm 3.1.1 in Figure 3.1). This algorithm uses the function $TRANS(\mathbf{x} \rightarrow \mathbf{y}, f)$ which is defined to transform the function f on the variables x_i to a function on the variables y_i.

In our SPDOR example, let us consider the excitation region for $x\downarrow$. In the naive graph shown in Figure 3.2(a), this region is just $\{(00F1)\}$. In the "haunted" version shown in Figure 3.2(b), it is extended to $\{(RRF1), (0FF1), (F0F1)\}$. The quiescent set for the same transition is $\{(RR0F)\}$ ($\{(000F)\}$ in the naive derivation).

Algorithm 3.1.1 (Find Excitation Regions)
set_of_bdds FindER(bdd N,bdd ES) {
  Do {
    Pick $s \in ES$ *(at random)*
      $\rho = s$;
      Do {
        $ER_k = \rho$;
        $\rho = ER_k \vee exist_q(\mathbf{x}', ES(\mathbf{x}) \wedge ER_k(\mathbf{x}') \wedge N(\mathbf{x}, \mathbf{x}')) \vee$
            $TRANS(\mathbf{x}' \rightarrow \mathbf{x}, exist_q(\mathbf{x}, ER_k(\mathbf{x}) \wedge ES(\mathbf{x}') \wedge N(\mathbf{x}, \mathbf{x}')))$;
      } **While**$(ER_k \neq \rho)$;
    Add ER_k *to bdd set ER;*
    $ES = ES \wedge \neg ER_k$;
  } **While** (ES)
  return (ER)
}

Figure 3.1. Algorithm to find excitation regions.

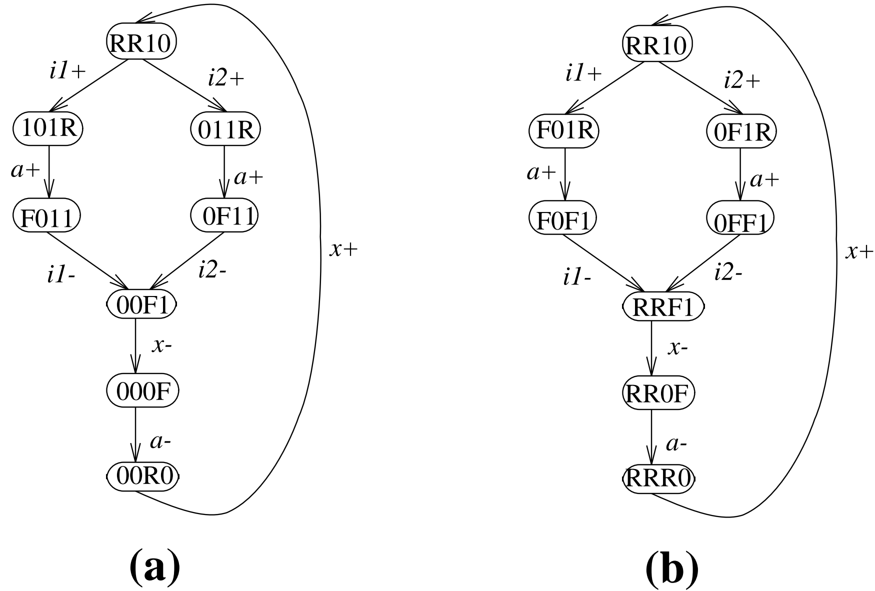


Figure 3.2. SPDOR graphs:(a)incorrect enablings, (b)correct enablings.

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 3.3). 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 3.3(a). It may also be created using a complex gate known as a *generalized C-element* (gC) [25]. Figure 3.3(b) shows a transistor-level gC design using a weak feedback staticizer, and Figure 3.3(c) shows a fully static design.

3.2.1 Single cube covers

In [11], a parametrized family of decompositions of high-fanin gates is investigated at one time by adding additional 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 will be ANDed together to produce a BDD which repre-

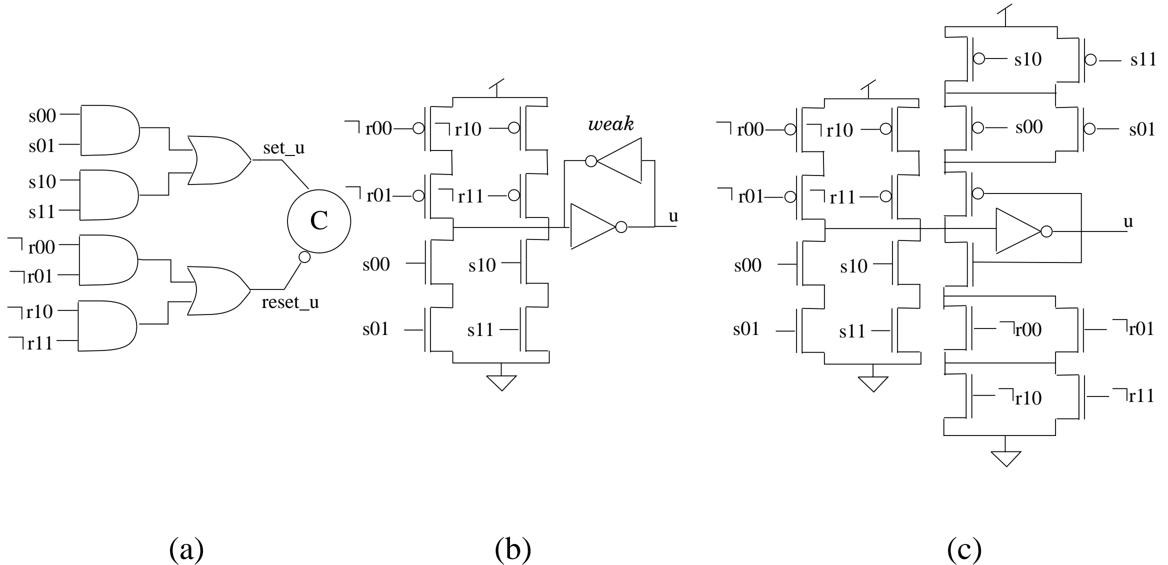


Figure 3.3. Circuit types:(a) a standard C-element design, (b) a generalized C-element design with weak-feedback, and (c) a fully-static gC design.

sents every possible potential single cube cover of the corresponding ER.

$$C_0(x*, k) = \bigwedge_i \Psi_{i,0}, \text{ where } \Psi_{i,0} = [(\chi_{i,0} \Rightarrow x_i) \wedge (\chi_{n+i,0} \Rightarrow \neg x_i)]$$

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.

3.2.2 Multicube covers

Occasionally an excitation region is found which cannot be covered by a single cube. An example is the RSG fragment shown in Figure 3.4. This is commonly known as a *nondistributive* region, since the excitation region has multiple minimal entry points. The ER for $c\uparrow$ is the set {(1RR),(R1R),(11R)}. The state (RR0), however, cannot be included, because c is stable low. Therefore, no single cube will describe the entire region. A possible solution is to add state variables to change the state coding [22]. Our approach is to create a SOP block to represent this region, instead of a simple "AND" gate design. 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 of the ER.

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

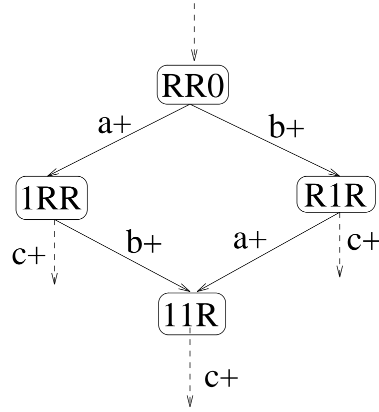


Figure 3.4. Fragment of an RSG for a standard OR gate.

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 [3] for speed-independent circuits and extended to timed circuits in [30].

3.3.1 gC cover violations

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 to say, 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.

3.3.2 SC cover violations

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

$$ER \subseteq C \cap \Phi \subseteq ER \cup QS.$$

That is to say that we 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: $V = V_1 \vee V_2$.

3.3.3 Correct covers

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 3.5 shows the resulting BDD for a SC implementation of the set region for signal c from Figure 3.4. Light arrows represent FALSE paths and dark arrows represent TRUE paths. p_a represents the χ variable representing a in the first cube, $pnot_a$ represents the χ variable representing $\neg a$ in the first cube, and $p_a 0$ represents the χ variable representing a in the second cube. There are six valid two-cube covers for this region: $a \vee b, (a \wedge \neg b) \vee b, a \vee (\neg a \wedge b), (a \wedge \neg c) \vee (b \wedge \neg c), (a \wedge \neg b \wedge \neg c) \vee (b \wedge \neg c), (a \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c)$.

Figure 3.6 shows the results for the feedback control signal x in the SPDOR example. The BDDs indicate that the pull-up stack can be enabled whenever a is false, and can be further restricted to only those states where some combination of x,i_1,and i_2 are false. Similarly, the pull-down stack must be on when a is high, but can be further restricted to those states where x is high. The resulting generalized C-element implementation is shown in Figure 3.7. Transistors shown in lighter print are optional. While this example shows some of the flexibility of the system, it should be noted that the end result is that this gate can be implemented with an inverter. As shown in the results section, other examples have many implementations of minimal size.

3.4 Results

The complete BDD timed circuit synthesis procedure shown in Algorithm 3.4.1 (see Figure 3.8) 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 3.1. The first column shows the number of gC style solutions that were found by the BDD synthesis procedure that can be implemented within the stack size limits. Most of the examples have a huge number of possible implementations with four of 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

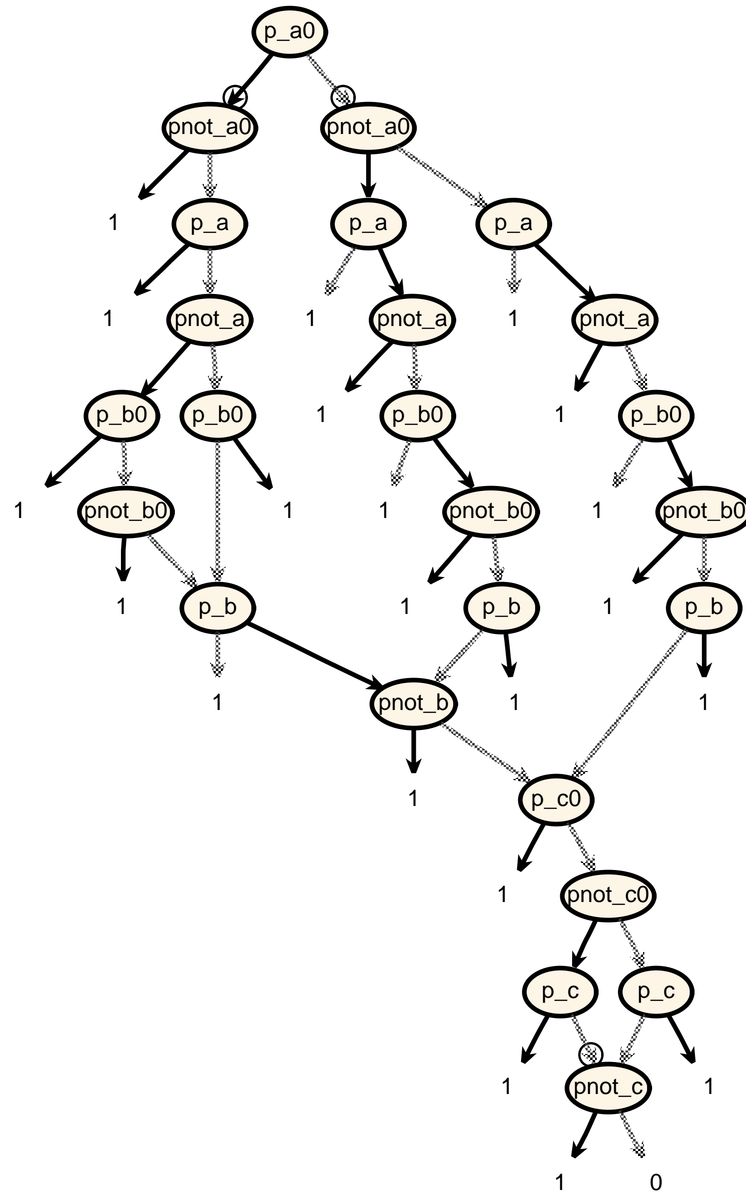


Figure 3.5. Valid cover for $c\uparrow$ for a standard OR gate.

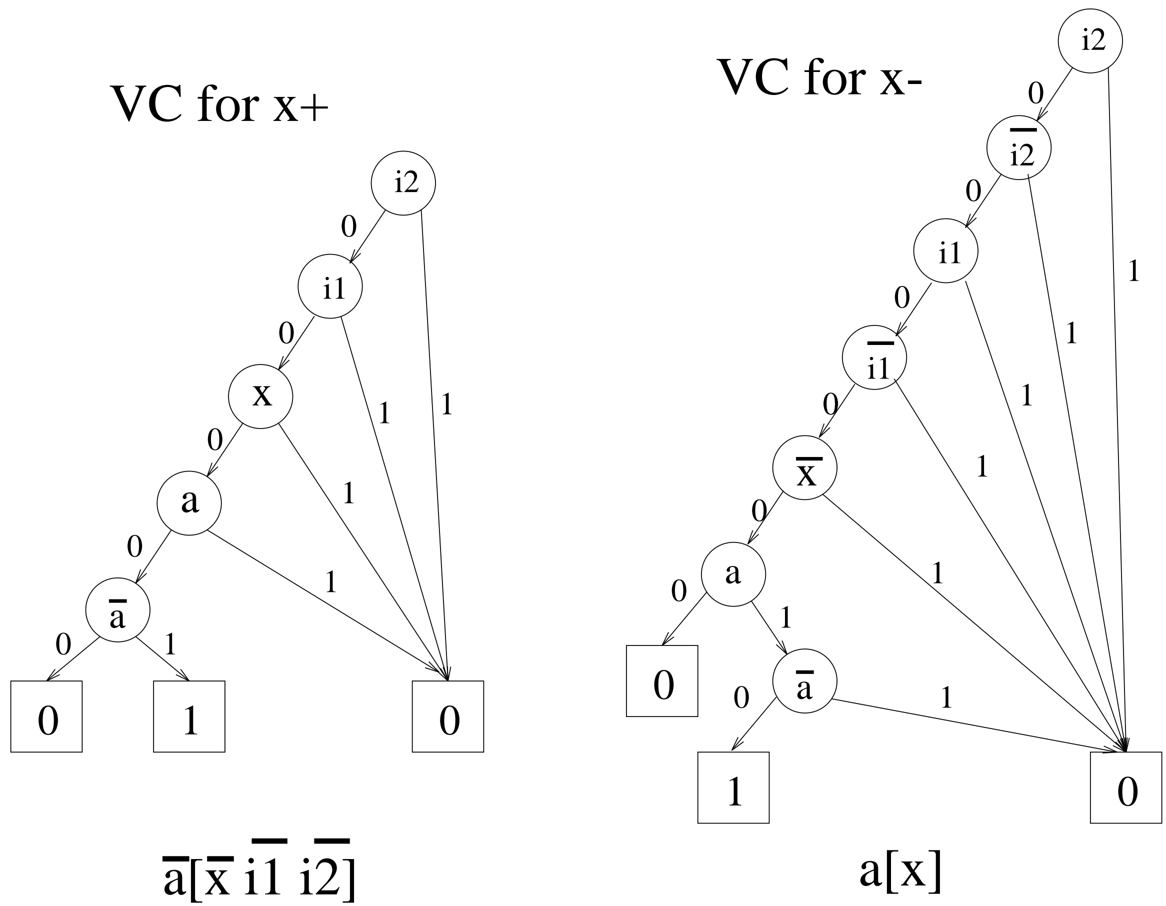


Figure 3.6. Valid cover for SPDOR feedback gate.

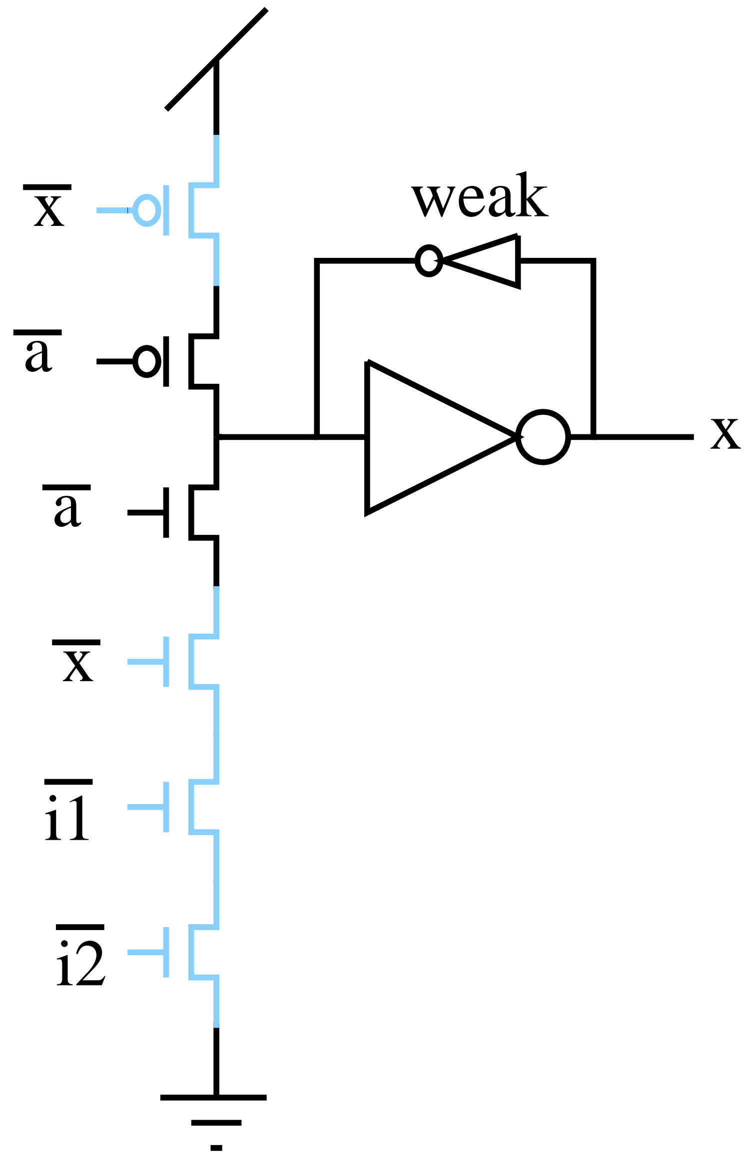


Figure 3.7. Valid circuit for SPDOR feedback gate.

Table 3.1. Experimental results.

	# of Solutions		Synthesis Time		
Examples	< 4	min	Implicit	Single-Cube	General
spdor	8192	1	.011	.0068	.35
spdand	512	1	.010	.0058	.28
cnt	614656	1	.043	.096	1.03
mmuoptSV	1.3×10^{23}	405	.44	.15	1.7
mmuopt	3.7×10^{9}	4	.089	.029	.65
slatch	1.3×10^{15}	2	.10	.022	.83
elatch	9.4×10^{12}	4	.13	.026	.85
SELopt	need decomp	4	.53	.082	1.5
tsbm	need decomp	4	5.0	FAIL	3.3
scsiSVT	3.2×10^{9}	18	.037	.034	.67
lapb	16384	1	.023	.0098	.43
lapb2	1.2×10^{9}	1	.26	.078	1.1
lapb3	6.1×10^{15}	2	2.1	1.6	20
lapb4	$2,2 \times 10^{22}$	4	13.4	8.4	29
fifo	1.7×10^{11}	4	.17	.048	.81
fifo2	1.9×10^{27}	16	1.8	.5	3.9
fifo3	2.1×10^{43}	64	2.2	9.1	53

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 many, most notably *eager* with 2304, have multiple minimal solutions which will not be 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. Filters have been employed to reduce the set to those having reasonable implementations in CMOS technology (i.e., those implementations which require transistor stacks of four or less.) It is interesting to note that often, as in the case of the SPDOR gate, all valid implementations are within the allowed stack size. 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 implementations in parallel.

The final three entries in the table are the runtimes to synthesize these circuits using the implicit approach, a heuristic single cube approach, and an exact multicube approach. Although somewhat slower than the heuristic single cube algorithm, the BDD synthesis method never fails, and in comparable runtime, finds BDD representations for a large number of possible synthesis solutions. The heuristic algorithm fails when multicube covers are required. The BDD method typically takes more than an order of magnitude less time than the general algorithm while still finding large numbers of solutions.

Algorithm 3.4.1 (Synthesize)

```
bdd_list Synthesize(RSG G) {
  S = FindStateGraphBDD(G);
  G = Haunt(G);
  N = NextState(G);
  Foreach x_i in x{
    Foreach x_i* in {x_i ↑, x_i ↓}{
      C = Generate C_0;
      If (* =↑){
        QS(x_i*) = exist_q(x', x_i ∧ x'_i ∧ N);
        ES(x_i*) = exist_q(x', ¬x_i ∧ x'_i ∧ N);
      } else{
        QS(x_i*) = exist_q(x', ¬x_i ∧ ¬x'_i ∧ N);
        ES(x_i*) = exist_q(x', x_i ∧ ¬x'_i ∧ N);
      }
      Foreach ER_k in ER{
        Do {
          if (gC) then
            V = S ∧ ¬ES ∧ ¬QS;
          else{
            V_1 = S ∧ ¬ER_k ∧ ¬QS;
            V_2 = TRANS(x' → x, exist_q((x, N(x, x')∧¬C(x)∧C(x')∧¬ER_k(x')));
            V = V_1 ∨ V_2;
          }
          VC = univ_q(x, (¬C ∨ ¬V) ∧ (¬ER_k ∨ C));
          If (VC = ∅)
            C = C ∨ Generate next C_l;
        }While (VC = ∅);
        add VC to set_of_results;
      }
    }
  }
  Return set_of_results
}
```

Figure 3.8. Function to synthesize circuit from a reduced state graph G

CHAPTER 4

CONCLUSIONS AND FUTURE WORK

One never notices what has been done; one can only see what remains to be done.
- Marie Curie, letter(1894)

This thesis presents a new implicit synthesis method for timed circuits which utilizes BDD based algorithms and data structures to allow the synthesis of larger timed circuit implementations. We formulated a MTBDD representations to represent 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).

Although this algorithm has led to a substantial reduction in memory usage, this has come at the cost of longer running times. We believe that this is not inherent in the methodology and plan to explore ways of optimizing our implementation. We would also like to extend the use of BDDs to other data structures in our algorithm. Specifically, the stack used during state space exploration continues to consume large amounts of memory. We attempted to synthesize *lapb5* and *fifo4*, but exhausted the available memory rapidly due to explosive stack growth. Storing the stack entries implicitly will hopefully reduce the size of the individual stack frames, allowing even larger specifications to be explored. Finally, we would like to research variable orderings and their affect on sharing, including possible methods of reordering and heuristics for static orderings.

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 [11, 18] to timed circuits.

REFERENCES

[1] Alur, R. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, August 1991.

[2] Beerel, P., and Meng, T.-Y. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 581–587.

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

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

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

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

[7] Brunvand, E., and Sproull, R. F. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1989), IEEE Computer Society Press, pp. 262–265.

[8] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers 35*, 8 (Aug. 1986), 677–691.

[9] Bryant, R. E. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer-Aided Design* (1995), IEEE Computer Society Press.

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

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

[12] Chu, T.-A. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

[13] Chu, T.-A., and Glasser, L. A. Synthesis of self-timed control circuits form graphs: An example. In *Proc. International Conf. Computer Design (ICCD)* (1986), IEEE Computer Society Press, pp. 565–571.

[14] Chung, E., and Kleeman, L. An optimal approach to implementing self-timed logic circuits from signal transition graphs. *Australian Telecommunications Research 27*, 2 (1993), 41–56.

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

[16] Clarke, E., MacMillan, K., Zhao, X., Fujita, M., and Yang, J.-Y. Spectral transforms for large boolean functions with application to technology mapping. In *30th Design Automation Conference* (June 1993), pp. 54–60.

[17] Coates, B., Davis, A., and Stevens, K. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal 15*, 3 (Oct. 1993), 341–366.

[18] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., and Yakovlev, A. Technology mapping of speed-independent circuits based on combinational decomposition and resynthesis. In *Proc. European Design and Test Conference* (1997), pp. 98–105.

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

[20] Ebergen, J. C. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[21] Hauck, S. Asynchronous design methodologies: An overview. Tech. Rep. TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[22] Kondratyev, A., Kishinevsky, M., Lin, B., Vanbekbergen, P., and Yakovlev, A. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference* (June 1994), pp. 56–62.

[23] Lavagno, L. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs.* PhD thesis, U.C. Berkeley, Nov. 1992. Technical report UCB/ERL M92/140.

[24] Lewis, H. R. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Tech. rep., Harvard University, July 1989.

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

[26] MENG, T. H.-Y., BRODERSEN, R. W., AND MESSERSCHMITT, D. G. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design 8*, 11 (Nov. 1989), 1185–1205.

[27] MOLNAR, C. E., FANG, T.-P., AND ROSENBERGER, F. U. Synthesis of delay-insensitive modules. In *1985 Chapel Hill Conference on Very Large Scale Integration* (1985), H. Fuchs, Ed., Computer Science Press, pp. 67–86.

[28] MOLNAR, C. E., JONES, I. W., COATES, B., AND LEXAU, J. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), IEEE Computer Society Press.

[29] MYERS, C. J. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[30] MYERS, C. J., ROKICKI, T. G., AND MENG, T. H.-Y. Automatic synthesis of gate-level timed circuits with choice. In *Proc. 16th Conf. on Advanced Research in VLSI* (1995), IEEE Computer Society Press, pp. 42–58.

[31] NOWICK, S. M., AND DILL, D. L. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1991), IEEE Computer Society Press, pp. 318–321.

[32] PASTOR, E., AND CORTADELLA, J. Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1993), IEEE Computer Society Press, pp. 250–254.

[33] PURI, R., AND GU, J. Asynchronous circuit synthesis; persistency and complete state coding constraints in signal transition graphs. *Int. Journal Electronics 75*, 5 (1993), 933–940.

[34] R.I.BAHAR, FROHM, E. A., GAONA, C. M., HACHTEL, G., MACII, E., PARDO, A., AND SOMENZI, F. Algebraic decision diagrams and their applications. In *International Conference on Computer Design* (Nov. 1993), IEEE, pp. 188–191.

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

[36] ROKICKI, T. G., AND MYERS, C. J. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verification* (1994), Springer-Verlag, pp. 468–480.

[37] Unger, S. H. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[38] Vanbekbergen, P., Lin, B., Goossens, G., and de Man, H. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 112–117.

[39] Yakovlev, A. V., Petrov, A., and Rosenblum, L. Synthesis of asynchronous control circuits from symbolic signal transition graphs. In *Asynchronous Design Methodologies* (1993), S. Furber and M. Edwards, Eds., vol. A-28 of *IFIP Transactions*, Elsevier Science Publishers, pp. 71–85.

[40] Yun, K. Y., Dill, D. L., and Nowick, S. M. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1992), IEEE Computer Society Press, pp. 346–350.

[41] Zheng, H. Specification and compilation of timed systems. Master's thesis, University of Utah, 1998.

View publication stats