

**ALGORITHMS FOR SYNTHESIS AND
VERIFICATION OF TIMED CIRCUITS AND
SYSTEMS**

by

Wendy A. Belluomini

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

Doctor of Philosophy

Department of Computer Science

The University of Utah

September 1999

Copyright © Wendy A. Belluomini 1999

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Wendy A. Belluomini

This dissertation 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

Erik Brunvand

Al Davis

Steven Burns

Ganesh Gopalakrishnan

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Wendy A. Belluomini 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 Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

In order to increase performance, circuit designers are beginning to move away from traditional, synchronous designs based on static logic. Recent design examples have shown that significant performance gains are realized when aggressive circuit styles are used. Circuit correctness in these aggressive circuit styles is highly timing dependent, and in industry they are typically designed by hand. In order to automate the process of designing and verifying timed circuits, algorithms to explore the reachable state space of the circuit under the timing constraints are necessary.

This thesis presents a new specification method for timed circuits, timed event/level (TEL) structures, and new algorithms for exploring a timed state space. The TEL structure specification allows the designer to specify behavior controlled by signal transitions, which is best for representing sequencing, and behavior controlled by signal levels, which is best for representing gate level circuits. This thesis also presents algorithms based on partially ordered sets (POSETs) that explores the timed state space of the TEL structure. Results using the new specification method and algorithms show orders of magnitude improvement over previous techniques in both speed and memory performance. The algorithms have also been successfully applied to several circuit examples from the recently published experimental Gigahertz processor developed at IBM. The speed and memory performance improvements of the algorithm allow automatic synthesis and verification of complex timed circuits, making them an attractive design alternative.

To Areal, who purred through the math.

CONTENTS

ABSTRACT	iv
LIST OF FIGURES	viii
LIST OF TABLES	x
NOTATION AND SYMBOLS	xi
ACKNOWLEDGEMENTS	xiii
CHAPTERS	
1. INTRODUCTION	1
1.1 Previous Work	3
1.1.1 Circuit Specification Approaches	3
1.1.2 Time Separation of Events Algorithms	5
1.1.3 Timed State Space Exploration Algorithms	6
1.1.4 State Space Reduction	7
1.2 Contributions	8
1.3 Thesis Overview	10
2. TIMED EVENT/LEVEL STRUCTURES	11
2.1 Motivating Example	12
2.2 The semantics of TEL structures	17
2.2.1 Timed event/level structures	17
2.2.2 Examples	19
2.2.3 Timed Firing Sequences	23
2.2.4 Examples of Firing Sequences	27
2.3 Summary	29
3. GEOMETRIC TIMING ANALYSIS	30
3.1 Defining Equivalence Classes	30
3.2 Timed State Space Exploration	32
3.3 Representing Time	36
3.4 Examples	39
3.5 Summary	43
4. POSET TIMING I	44
4.1 Creating Larger Equivalence Classes	44
4.2 POSET Algorithm	52
4.3 Example	55
4.4 Summary	57

4.5	Appendix	58
5.	POSET TIMING II	66
5.1	Extending the Required Set	66
5.2	Adding Reordering Restrictions	68
5.3	Simplified Restrictions	71
5.4	Extended POSET Algorithm	73
5.5	Example	76
5.6	Summary	79
6.	OPTIMIZATIONS	80
6.1	Subsets	80
6.2	Supersets	81
6.3	Untimed Rule Optimization	82
6.4	Merge	82
6.5	Interleaving	84
6.6	Summary	89
7.	IMPLICIT METHODS	90
7.1	Representing Geometric Regions	90
7.2	Representing the Reduced State Graph	95
7.3	Summary	97
8.	VERIFICATION	98
8.1	Constraint Rules	99
8.2	Success and Failure Sequences	100
8.3	Checking for Failures	102
8.4	Example	103
8.5	Summary	106
9.	RESULTS	107
9.1	Comparison with <code>Orbits</code>	108
9.2	Comparison with Other Verification Methods	117
9.3	Implicit Methods	123
9.4	Application to Synchronous Circuits	125
9.4.1	Verification of Gate Level Models	127
9.4.2	Verification of Abstracted Models	129
9.5	Summary	133
9.6	Appendix - Reproducibility	134
10.	CONCLUSIONS AND FUTURE WORK	138
10.1	Summary	138
10.2	Future Work	139
10.2.1	Specification	139
10.2.2	Algorithms	140
10.2.3	Applications	141
	REFERENCES	143

LIST OF FIGURES

2.1	Specifications for <i>sbuf-send-pk2</i> controller.	12
2.2	Complete Petri net for <i>sbuf-send-pk2</i> controller.	13
2.3	TEL structure for <i>sbuf-send-pk2</i> controller circuit.	14
2.4	TEL structure for <i>sbuf-send-pk2</i> controller environment.	14
2.5	Handshaking expansion for <i>sbuf-send-pk2</i> controller circuit.	15
2.6	Examples of TEL structures.	19
2.7	A delayed-reset domino gate.	20
2.8	TEL structure for the gate in Fig. 2.7 and its environment.	21
2.9	VHDL description of the domino gate	22
2.10	Conflict behavior.	26
3.1	Timed state space exploration.	34
3.2	Find timed enabled rules.	35
3.3	(a) TEL structure with multiple behavior rules, and (b) non-convex region.	38
3.4	Procedure for updating the geometric region.	38
3.5	Firing rules.	40
3.6	Firing rules with levels.	42
4.1	A choice computation.	51
4.2	Update the geometric region using POSETs.	54
4.3	Procedure for updating the geometric region.	54
4.4	Example of timing with partially ordered sets.	56
5.1	Transformations to simple and and or	71
5.2	Procedure for updating the POSET matrix.	74
5.3	Example for POSET algorithm with levels.	77
6.1	Region matrices.	81
6.2	Merges.	82
6.3	Procedure for matching two rule sets.	84
6.4	Example of interleaving optimization.	85
6.5	POSET matrices with various causal places.	86

6.6	TEL structure where p4 cannot be created last.	87
6.7	A restricted POSET matrix.	87
7.1	Function to extract a BDD for the rule set R_m	91
7.2	MTBDD representation of (a) R_m , (b) the number “2”, and (c) a geometric region matrix.	92
7.3	Function to create a MTBDD for the matrix M	93
7.4	MTBDD representation of a timed state.	94
7.5	A false-terminated array holding (a)one, (b)two, or (c)three matrices.	96
7.6	(a) A reduced state graph (RSG), (b) a BDD for the state RR0, and (c) the characteristic function BDD (S) for the state space.	96
8.1	Example of a constraint rule	99
8.2	Check for deadlock.	102
8.3	Find timed enabled rules.	103
8.4	Update the region and check constraints.	104
8.5	Example of new update algorithm.	105
9.1	TEL structure for a 2-bit counter.	108
9.2	TEL structure for one LAPB stage.	111
9.3	2 level selector.	112
9.4	TEL structure for one selector unit.	113
9.5	The tag unit circuit.	115
9.6	TEL structures for the Alpha and Beta examples.	118
9.7	Comparative performance for the Alpha example.	119
9.8	Comparative performance for the Beta example.	120
9.9	TEL structures for the STARI example (a) the clock process with timing constraints = $[\pi, \pi]$; (b) the transmitter process and (c) the receiver process with timing constraints = $[0, skew]$; and (d) a STARI FIFO stage with timing constraints = $[l, u]$	121
9.10	Stari results with POSETs and with COSPAN.	122
9.11	FIFO memory performance.	124
9.12	The MLE circuit.	128
9.13	PLA control.	128
9.14	Model for the compare unit.	129
9.15	MFXU structure.	132
9.16	The CLZ circuit.	133

LIST OF TABLES

9.1 Results for counters.	110
9.2 Results for the LAPBs.	111
9.3 Results for the selector unit	114
9.4 Results for tagunit.	116
9.5 Comparison with level based tag unit	117
9.6 Location of examples.	135
9.7 Location of examples - continued.	136
9.8 Location of examples - continued.	137

NOTATION AND SYMBOLS

*The good Christian should beware of mathematicians, and all those who make empty prophecies. The danger already exists that the mathematicians have made a covenant with the devil to darken the spirit and to confine man in the bonds of Hell.
- St. Augustine*

Symbol	Meaning	Reference
A	Set of actions in a TEL structure.	Definition 2.2.1
C	Set of constraint rules in a TEL structure.	Section 8.1
context firing	A firing that is necessary to satisfy a boolean expression.	Definition 5.1.2
$causal(\sigma_i, \sigma_j, \sigma)$	Event firing σ_i is causal to event firing σ_j .	Definition 4.1.2
$choice_set(r)$	Set of events whose firing disables r .	Definition 2.2.1
E	Set of events in a TEL structure.	Definition 2.2.1
$E_c(\sigma_i, \sigma_{0..n})$	Causal event function.	Definition 2.2.6
$E_m(r, \sigma_{0..n})$	Returns the event whose firing enabled r in $\sigma_{0..n}$.	Definition 3.3.1
$enabled(\sigma_{0..n})$	The set of rules enabled by firing sequence $\sigma_{0..n}$.	Definition 2.2.2
F	The set of failure sequences.	Section 8.2
$firable(\sigma)$	Set of events which can fire.	Definition 2.2.4
$L(\sigma_i)$	The rule or event fired by σ_i .	Section 2.2.3
M	The constraint (or geometric region) matrix.	Section 3.2
$max_advance(\sigma_{0..n}, \tau)$	Amount of time before a clock exceeds its bound.	Definition 3.1.2
N	Set of signals in a TEL structure.	Definition 2.2.1.
PM	The POSET matrix.	Section 4.2
R	Set of rules in a TEL structure.	Definition 2.2.1
R_0	Set of initially marked rules in a TEL structure.	Definition 2.2.1
R_{en}	The set of enabled rules.	Section 3.2
R_f	The set of fired rules.	Section 3.2
R_m	The set of marked rules.	Section 3.2
$required(\sigma_i, \sigma_{0..n})$	The set of firings in $\sigma_{0..n}$ that are necessary for σ_i to fire.	Definition 4.1.3

Symbol	Meaning	Reference
S	The set of success sequences.	Section 8.2
s_0	Initial state of a TEL structure.	Definition 2.2.1
s_c	The current state.	Section 3.2
$\#$	Set of conflicts in a TEL structure.	Definition 2.2.1
σ	A firing sequence.	Section 2.2.3
σ_i	Firing i in firing sequence σ .	Section 2.2.3
Σ	The set of all reachable firing sequences.	Section 2.2.3
τ	A timing assignment.	Definition 2.2.7
$\phi(\sigma)$	Returns boolean state after executing σ .	Section 2.2.3
Φ	The set of reachable boolean states.	Section 3.2
Γ	The set of transitions between boolean states.	Section 3.2
$\rho(\sigma)$	A firing sequence constructed by changing the firing order of σ .	Section 4.1
$\rho(\sigma_i)$	The index of firing σ_i in the new sequence constructed by ρ .	Section 4.1

ACKNOWLEDGEMENTS

This thesis is made possible by the support, encouragement and technical input of my three advisors. My interest in asynchronous design began when I took Alain Martin's class at Caltech. I would like to thank him for taking the time to involve me in research as an undergraduate. Without that experience, I most likely would not have gone to graduate school. Next, I would like to thank Steve Burns, my advisor at the University of Washington, for giving me the time and freedom to discover the research topic that I wanted to explore. Finally, I would like to thank Chris Myers, my advisor at the University of Utah. My years with Chris have been very productive, and his guidance has made it possible for me to complete my degree at Utah in three years. I would especially like to thank Chris for his support through numerous personal and professional "crises".

A number of other people have contributed ideas that improved this work. I would like to thank my committee members Al Davis, Ganesh Gopalakrishnan, and Erik Brunvand for their input during my time at Utah and for their helpful comments on this thesis. I would also like to thank Peter Beerel for his help in setting up the theory. The chapter on implicit methods is joint work with Robert Thacker and I enjoyed my collaboration with him. The work in this thesis builds on work by Tom Rokicki and I would like to thank Tom for writing a wonderfully readable thesis and for answering questions regarding its contents. Finally, I thank Peter Hofstee for giving me the chance to apply my work to real, industrial circuits.

The support I received from friends and family was essential in completing this work. I would like to thank Dan Fasulo and Sean Sandees for their friendship at the University of Washington and for being there for me when Steve decided to leave the university. I would like to thank my friends and officemates at the University of Utah, Robert Thacker and Eric Mercer, for always being ready to take a break and kill some time in the office. Coleen Hoopes has my eternal thanks for helping to shelter me somewhat from the bureaucracy of the university. Finally I would like to thank my parents, Frank and Anita Belluomini, and my boyfriend, Chandler McDowell, for their love and support throughout the past

five years that I have been in graduate school. They were always there to encourage me when things were going badly and to celebrate when things were going well, even if it was over a long distance line.

CHAPTER 1

INTRODUCTION

*Time is a great teacher, but unfortunately it
kills all its pupils.*
- Hector Berlioz

In order to increase performance, circuit designers are beginning to move away from traditional, synchronous designs based on static logic. Recent designs, such as the Intel RAPPID instruction length decoder described by Rotem in [60] and the IBM guTS microprocessor described by Hofstee in [35] have shown that large performance gains can be realized using aggressive circuit styles which make many timing assumptions. The RAPPID chip is an asynchronous implementation of an instruction length decoder for a Pentium II instruction set. It achieves a 300% performance improvement while dissipating half the power of the synchronous implementation on the same process. The guTS microprocessor is a synchronous implementation of a Power PC instruction set running at 1 Gigahertz on a 0.25μ CMOS process available in 1997. Although both designs achieve significant performance gains, they are experimental designs. Many obstacles need to be overcome before the circuit styles developed in these designs can be used in production. One of the main obstacles is the lack of design automation for timed design styles.

Design of efficient timed circuits requires timing information to be used throughout the synthesis and verification loop. The synthesis process begins with a specification of circuit behavior that includes any timing assumptions the designer wishes to make. Although the designer cannot make precise timing assumptions at the specification stage, he does know something about the timing behavior of the circuit and timing relationships between signal transitions can be bounded. In order to generate a circuit for this specification, the synthesis tool then finds its reachable state space. Even if the specification contains no timing information, this would be an exponential problem, and adding timing information can complicate the problem further. However, if timing information is represented well

it can sometimes designate large portions of the state space as unreachable and therefore reduce the time it takes to generate the reachable state space and synthesize the circuit.

Once the state space is found, it is used by an algorithm presented by Myers in [50] to synthesize a gate level timed circuit. In this algorithm a function block for each output signal, consisting of a C-element with a sum of products block for the set and another for the reset. Each “product” block implements a single excitation region for a given output signal. 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. The algorithm also determines set of excited states, which is the union of the excitation regions for a given signal transition and an associated set of stable, or quiescent, states. For a rising transition $x+$, this is the set of states where the signal x is stable high, and is similarly defined for a falling transition. The algorithm then uses these sets of states to set up and solve a covering problem whose constraints require that the resulting circuit is hazard-free.

Once the physical design for the gate level circuit produced by the synthesis algorithm is complete, the circuit must formally verified. Timing assumptions in the specification are made before there is any data available on the physical behavior of the circuit. Therefore, these assumptions must be checked once the circuit is synthesized to make sure the timing behavior of the implementation is consistent with the specification.

This thesis develops the algorithms necessary to use timing information in both synthesis and verification. It first describes a specification method which is designed specifically for circuits. It then presents an extension of the standard geometric region (or DBM) method of timing analysis that is capable of analyzing these specifications. Although this method works well for some specifications, it can suffer from state space explosion when applied to highly concurrent examples. Therefore, we present a new algorithm, called partially ordered set (POSET) timing which can reduce the state space size by orders of magnitude. To further improve the speed of the algorithm, many optimizations are made to deal with special cases that can eliminate many states. Finally, a formal framework for verification using the algorithm is presented. The synthesis and verification algorithms are implemented in the CAD tool *ATACS* and are applied to many examples, including high performance synchronous circuits from a large industrial example. This indicates that the algorithm not only produces significantly improved results when applied to academic benchmarks, but can also be useful to industry in the design of future generation, high-performance timed circuits.

1.1 Previous Work

The development of algorithms to synthesize and analyze asynchronous circuits has been a very active area of research. However most of this work has been directed toward untimed design styles. These styles, while robust, require a large amount of hand optimization in order to produce competitive performance. There has also been extensive work in the area of verification of timed systems. This work can be applied to timed circuits, however, approaches specifically designed for circuits lead to better results and automation.

1.1.1 Circuit Specification Approaches

The difficulty of circuit synthesis depends heavily on the type of specifications that are allowed. Generally, very restrictive specification approaches make synthesis easier, but are not useful for large, complex designs. Restrictive specification approaches may also result in slower circuits, since designers cannot specify many of the optimizations they would like to make. More flexible and expressive specification methods make synthesis harder, but allow faster and more complex circuits to be synthesized. In order for an asynchronous specification method to allow the synthesis of fast, complex designs, it must have good support for concurrency and timing information, and it must be able to specify behavior based on both signal transitions and signal levels.

There are currently two general approaches to specifying the behavior of asynchronous circuits: language-based approaches and graph-based approaches. The two specification methods each allow a somewhat different class of circuit to be specified and require different methods for synthesis. Therefore, the specification method chosen can determine to a large extent the quality of the resulting circuit. Synthesis methods for language-based specifications directly translate a program into a circuit. One approach to this, proposed by van Berkel in [68] and Brunvand in [15], is syntax directed translation where language constructs are mapped directly to library blocks. In this method, signal levels and concurrency are supported, but timing information cannot be specified. Also, the circuits produced tend to be redundant and slow since optimizations are not seen when simply mapping program constructs to circuit blocks. In another language-based method, which is presented by Martin in [44], the specification program is translated to a circuit using a series of semantic preserving transformations. This approach also supports levels, but it requires a large amount of human intervention to be effective and has no support

for timing.

Graph-based specification methods require a specification that is lower level than language based methods, but can make synthesis of efficient circuits easier. Many researchers, including Chu [22], Meng [47], Lin [43], Vanbekbergen [69], and Lavagno [41] use an interpreted Petri net or *STG* for specification. STGs are very good at expressing concurrency. However, the traditional STG synthesis methods restrict the types of choice allowed in the net, and they have no support for the specification of level information or timing assumptions. There is an extension to STGs developed by Moon [49] that does support levels, but it requires a restricted environment and synthesis algorithms for this extended specification are not presented. Additionally, in [70] Vanbekbergen presents extensions to STGs that support levels and timing in STGs. However, in this work algorithms for synthesizing timed STGs with levels are not presented. Another graph-based method, Kishinevsky's *change diagrams*, is similar to STGs but removes some of the restrictions by adding different types of arcs to the specification. These additional arcs allow more disjunctive behavior to be specified. However, change diagrams have no provision for timing information. Yun [74, 75], Nowick [54], and Coates [25] specify circuits using asynchronous state-machines, and perform synthesis using burst-mode techniques. The burst-mode method allows one signal level to be specified on each arc of the state machine. However, burst-mode synthesis requires the *fundamental-mode* assumption which states that when a state change occurs, all of the changing outputs are allowed to settle before any change in the input signals. This can sometimes require adding delay between the circuit and its environment so that the inputs to the circuit do not change before the outputs settle. Also, state-machine based specification is not well-suited to expressing concurrency since state machines are inherently sequential. Finally, state machines do not express causality between output and input events directly, making it difficult to utilize timing assumptions to optimize the circuit.

The specification method used in the version of the **ATACS** tool described by Myers in [50] is a combination of the graph-based and language-based approaches. While the tool accepts language-based specifications as input, it does not directly use them for synthesis. Instead, **ATACS** compiles the input program into a graph, which is then used for synthesis. This version of **ATACS** uses *timed event-rule(ER) structures*, a variant of Winskel's event structures [72] with timing, for synthesis. Since timed ER structures separate causality from choice, they are both easier to generate from high-level descriptions, and easier to

analyze. Unlike all of the previously described specification methods, timed ER structures allow the use of explicit timing assumptions in synthesis. However, like STGs, timed ER structures have no support for levels in the specification. This can be quite limiting when trying to express things like true OR causality and many language constructs, such as conditional loops. This thesis presents a new specification method that adds levels to ER structures.

1.1.2 Time Separation of Events Algorithms

If a specification with timing information is used, a timing analysis step is necessary to find the timed state space. A number of timing analysis algorithms have been developed, and each is optimized to solve a different class of problem. A time separation of events algorithm is not designed explicitly for state space exploration. Its result is a minimum and maximum separation between two specified events. It can be used for state space exploration indirectly by calling it repeatedly during a state space exploration algorithm to determine which events are allowed to occur. However, none of the existing time separation of events algorithms are suitable for timed state space exploration of a sufficiently expressive specification.

In [46], Dill presents an algorithm for finding the minimum and maximum time separations between events in acyclic graphs. It is $O(n^3)$ in the number of events in the graph. This algorithm can be used for timed state space exploration if the specification graph is acyclic. However, most circuit specifications are cyclic.

In [51], Myers presents a polynomial time algorithm to compute an estimate of the minimum and maximum time differences between all events in a cyclic, choice-free graph. The algorithm works by unfolding the cyclic graph into an infinite acyclic graph and examining two finite acyclic subgraphs of the infinite graph to determine bounds on time differences between events. The estimate is usually sufficient for timed state space analysis and can be improved by analyzing larger subgraphs. The algorithm is $O(v \cdot e)$ where v is the number of vertices and e is the number of edges in the subgraph analyzed. The choice-free restriction is too limiting however, since most circuits need a choice semantics to represent non-deterministic behavior in the environment.

CTSE, presented by Hulgaard in [37, 36], provides a way to find a single exact time difference (separation) between two events in a cyclic graph including limited types of choice. This type of algorithm can be used for state space exploration by running the algorithm to determine the minimum and maximum time separation between every pair

of events in the specification. However, it has two drawbacks. The first is that the algorithm in [37, 36] is not able to analyze specifications with arbitrary choice and level based behavior. This drawback could be eliminated by the development of a more general algorithm. The second drawback is more fundamental. A time separation of events algorithm provides the minimum or maximum time separation between two events that is possible over all possible executions of the specification. A state space exploration algorithm needs to know the minimum and maximum time separation between events that can lead to a given boolean state. In many cases, certain boolean states are only reachable when the time operation between two events is less than its overall maximum or greater than its overall minimum. Since time separation of events algorithms lack state dependent timing information, a state space algorithm using this approach is approximate.

1.1.3 Timed State Space Exploration Algorithms

The first dividing factor between time state space exploration algorithms is how they represent time. The representation of the timing information has a huge impact on the growth of the state space. Timing behavior can either be modeled continuously (i.e., dense-time), where the timers in the system can take on any value between their lower and upper bounds, or discretely, where timers can only take on values that are multiples of a discretization constant. Discrete time has the advantage that the timing analysis technique is simpler and implicit techniques can be easily applied to improve performance as shown by Burch in [18] and Bozga in [13]. The worst case complexity of this approach is $O(|S|(k+1)^n)$ where S is the number of untimed states, n is the maximum number of places in the Petri net that can be marked, and k is the maximum value of any timing requirement. This worst case complexity is often approached in actual circuits and the state space explodes if the delay ranges are large and the discretization constant is set small enough to ensure exact exploration of the state space. For example a delay range of 117 to 269 has 153 discrete states if the discretization constant is set to one. Although the discretization constant can be larger than one if there is a larger number that divides all of the numbers used for delay ranges, this does not happen very often when delay numbers from actual circuit data are used.

Continuous time techniques eliminate the need for a discretization constant by breaking the infinite continuous timed state space into equivalence classes. All timing assignments within an equivalence class lead to the same behavior and do not need to

be explored separately. In order to reduce the size of the state space, the size of the equivalence classes should be as large as possible. In Alur’s *unit-cube* (or region) approach [1], timed states with the same integral clock values and a particular linear ordering of the fractional values of the clocks are considered equivalent. Although this approach eliminates the need to discretize time its complexity of $O(|S|^{\frac{n!}{\ln 2}}(\frac{k}{\ln 2})^n 4^{(1/k)})$ is much worse than discrete time and the state space using this method typically explodes if the delay ranges are large.

Dill [28], Berthomieu [12], Lewis [42], and Alur [3] present another approach to continuous time where the equivalence classes are represented as convex *geometric regions* (or zones). Geometric regions can be represented by sets of linear inequalities (also known as *difference bound matrices* or DBMs). The worst case complexity of this timing representation is worse than that of unit cube. However, the worst case complexity occurs less often when verifying real circuits. The larger equivalence classes generated by the geometric region method can often result in smaller state spaces than those generated by the unit-cube approach. The number of geometric regions can explode with geometric approaches since each untimed state has at least one geometric region associated with it for every firing sequence that can result in that state. In highly concurrent systems where many interleavings are possible, the number of geometric regions per untimed state is huge. In order for it to be effective, techniques are needed to reduce state space size.

1.1.4 State Space Reduction

A number of techniques have been proposed to deal with state explosion. Valmari [67], Godefroid [31] and McMillan [45] have proposed approaches that use stubborn sets [67], partial orders [31], or unfolding [45]. These techniques reduce the number of states explored by considering only a subset of the possible interleavings between events. They are targeted specifically at verification, and they allow the removal of interleavings since some interleavings are not relevant to the property that is being verified. These approaches have been successful, but they only deal with untimed verification.

The state space of timed systems is even larger than the state space of untimed systems and has been more difficult to reduce. Yoneda [73], Semenov [61], Verlind [71], and Bengtsson [11] have attacked this problem by reducing the number of interleavings explored using the partial order techniques developed for untimed systems. These algorithms compute a set of event firings that must be interleaved to ensure that the

desired property is checked. Any event firings not in the set are not interleaved. This reduces the state space significantly for highly concurrent specifications. While reducing the number of interleavings is useful, in [73, 61] one region is still required for every firing sequence explored to reach a state. If most interleavings need to be explored, these techniques could still result in state explosion. The algorithms from [71, 11] do address the problem of generating a unique region for every firing sequence. In [71] an algorithm which operates on timed Petri nets is proposed where transitions are given negative firing times in order to build regions that do not depend on the firing order. The work in [11] takes a related approach where the clocks associated with states in a timed automata are allowed to advance at different rates. However, since these techniques do not find the entire state space, they cannot be applied to synthesis. Logic synthesis algorithms for timed asynchronous circuits require that all of the boolean states allowed by the state space are found in order to create a correct logic implementation [52]. If the synthesis algorithm is given an incomplete state space, it cannot be guaranteed to generate logic that correctly responds to all inputs to the circuit.

Orbits, presented by Myers and Rokicki in [58, 59, 53], takes a somewhat different approach. It reduces the number of regions per untimed state by using *partially ordered sets* (or POSETs) of events rather than linear sequences to construct the geometric regions. Instead, the algorithm generates only one geometric region for any set of firing sequences that differ only in the firing order of concurrent events. This algorithm is shown in [59] to result in very few geometric regions per untimed state. This algorithm differs from the partial order approaches in that it still finds a complete state space and improvement achieved by **Orbits** is not dependent on the verification property. However, it is limited to specifications where the firing time of an event can only be controlled by a single predecessor event (known as the *single behavioral place (or rule) restriction*). In some cases, the single behavioral rule restriction can be worked around through transformations on the initial graphs [50], however the transformations cause a large increase in the complexity of the graphs which need to be analyzed. This thesis extends the algorithms presented in [58, 53] to work with much more flexible specifications.

1.2 Contributions

This thesis makes three main contributions to the area of synthesis and verification of timed circuits. The first contribution is in the area of specification. This thesis introduces

timed event/level(TEL) structures, an extension to the timed ER structures developed by Myers [50], which allows the general use of levels in the specification of a timed circuit. TEL structures allow information about levels to be included in the ER structure in the form of an arbitrary boolean expression. This provides a number of advantages over other specification methods. Since circuit behavior at its lowest level is based on the sampling of boolean values from wires, specifications with the ability to model this are much more compact than those based purely on signal events. Although purely event based specifications can model level-based circuit behavior, they are much larger, and thus they are more time consuming to analyze than level based specifications. Additionally, this direct correspondence facilitates translation of a circuit into a TEL structure both by compilation of a VHDL specification and by directly translating schematics by hand.

The next contribution is an algorithm to analyze the TEL structures efficiently. As mentioned earlier, this algorithm is an extension of the one introduced in by Rokicki in [58]. Although the algorithm does not improve the worst case complexity of the geometric approach, it reduces the size of the state space for most circuit specifications by eliminating the requirement that a new state must be generated by every possible interleaving of event firings. The algorithm does this by creating timed states based on a partial order created from the execution sequence being explored, instead of the total order of the sequence. It is therefore referred to as the partially ordered set, or POSET algorithm. The algorithm introduced in [58] is limited to specifications that meet the single behavioral rule requirement and have no level expressions. This thesis extends the algorithm to work on TEL structures and presents formal proofs for the correctness of the algorithm which are lacking in [58]. It also discusses a number of optimizations, including the use of implicit representations such as (MTBDDs) to further reduce the memory requirements of the state space exploration.

The final contribution is in the area of verification of high-performance timed synchronous and asynchronous circuits. Due to the synchronous abstraction, timing verification for synchronous circuits is seen as a much simpler problem than that of asynchronous circuits. However, in order to get the highest performance possible, synchronous designers are creating ever more aggressive circuit styles. These styles typically break the existing timing methodology and it takes a long time before the synchronous timing tools catch up, by which time designers have begun to work with an even more aggressively timed circuit style. The result of this lag between circuit design and CAD support is that aggressive

circuit styles are not used in production until long after they have been developed. This thesis shows how timing verification approaches developed for asynchronous circuits can also be used to verify timed synchronous circuits. Although the asynchronous algorithms have much greater complexity than the algorithms used for synchronous circuits, they have the advantage that they can be applied to any type of circuit style without any adaptations. This may help reduce the lag between the development of experimental circuit styles and their actual use.

1.3 Thesis Overview

The thesis is organized as follows: Chapter 2 introduces TEL structures. Chapter 3 describes the changes made to the basic geometric algorithm to deal with multiple behavioral rules and level expressions. Chapters 4 and 5 describe the POSET algorithm and its proof of correctness. Chapters 6 and 7 describe the optimizations that have been made to the algorithm to increase speed and memory performance. Chapter 8 discusses how verification properties are specified and checked. Chapter 9 presents results, and finally, Chapter 10 discusses conclusions and future work.

CHAPTER 2

TIMED EVENT/LEVEL STRUCTURES

The mere formulation of a problem is far more essential than its solution, which may be merely a matter of mathematical or experimental skills.

- Albert Einstein

As discussed in the previous chapter, most existing asynchronous CAD tools have one of two major weaknesses: they do not support explicit timing and they are purely event based. Timing assumptions can often make the difference between an asynchronous circuit that is faster than the corresponding synchronous circuit and one that is slower. Timing assumptions can be made manually by the designer, but this is very error prone. The lack of the ability to specify signal levels limits the usefulness of the tool and makes it difficult to specify any behavior where sampling the value of a signal is necessary. Simple concepts, such as a loop on a condition, often have complex or imprecise specifications if level information cannot be included. This makes asynchronous design tools harder to use and less appealing to industrial designers.

This chapter describes *timed event/level (TEL) structures*, which we first present in [8]. TEL structures are an extension to timed ER structures which allow the general use of levels in the specification of a timed circuit. Information about levels is included in the ER structure in the form of an arbitrary boolean expression. This makes it possible to extend the specification languages accepted by ATACS to allow the specification of conditional loops and true OR causality, as well as all other constructs that require waits on boolean expressions. TEL structures can be analyzed using a modified version of the geometric timing analysis method we presented in [9]. The faster POSET algorithm, which we presented in [7] is also capable of analyzing TEL structures without adding significant overhead. Therefore, TEL structures facilitate more general specifications without decreasing synthesis performance.

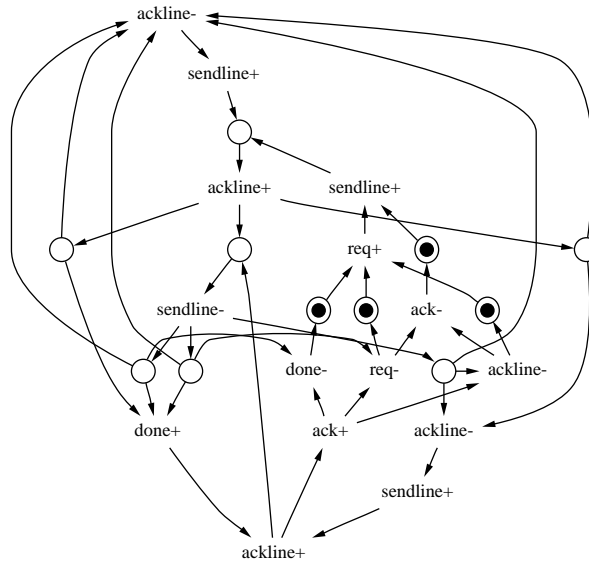


Figure 2.2. Complete Petri net for *sbuf-send-pk2* controller.

the packet transfer is over. The receiver then lowers *req*, *ackline*, and *done* in parallel and the sender, in response to this, lowers *ack*. This is a reasonably simple specification since the extended burst-mode machine can sample the level of the signal *done* when choosing whether to transition to state 2 or state 3 from state 1. It also appears simple since in burst-mode the environment is implicit and does not need to be expressed in the specification. Although this simplifies the specification, it makes it impossible to make detailed timing assumptions about the behavior of the environment. Figure 2.1(b) shows a free choice STG that is used to specify this circuit in the SIS benchmark suite. It is not very complex, but it is not complete. This STG only allows the transition *done+* to occur between the transitions *sendline-* and *ack+*. In the description of the circuit, however, *done+* can occur any time between a falling transition of *sendline* and a rising transition of *ackline*. It is necessary to restrict the behavior of *done* in order to express this circuit as a free choice STG. The Petri net that specifies the full behavior of the circuit, shown in Figure 2.2 is quite complex, and it does not have the free-choice property. This net was derived from a state graph of the circuit using *Petrify* [26] and would be very difficult to generate correctly either by hand or by compilation from a higher level language.

Figure 2.3 shows the TEL structure that represents the circuit for *sbuf-send-pk2* and


```

process circuit
*[[req]; sendline+; [¬done ∧ ackline → sendline−; [¬ackline]; sendline+; *
    | done ∧ ackline → (ack+ || sendline−); [¬req ∧ ¬ackline]; ack−;]]
endprocess

process environment
*[req+; [sendline]; ackline+;
    [ ¬sendline → (done+ || ackline−); [sendline]; ackline + [¬sendline ∧ ack];
      (req− || ackline− || done−); [¬ack]
    | ¬sendline → ackline−; [sendline]; ackline+; *]]
endprocess

```

Figure 2.5. Handshaking expansion for *sbuf-send-pk2* controller circuit.

wait until the expression it contains is satisfied. Waits can also be placed in guarded commands, for example, $[a \rightarrow b+ \mid c \rightarrow d+]$. This guarded command chooses between executing $b+$ and $d+$ depending on whether a or c is true. The handshaking semantics defined in [44] require that guards on guarded commands must be mutually exclusive. The $*$ operator indicates a loop. If it used at the end of a guarded command it indicates that control returns to the beginning of the guarded command. If it is used at the beginning of a process, it indicates that the process repeats forever.

The first thing to notice about the TEL structure which represents the handshaking expansion is that each process in the specification is represented with a separate TEL structure. In this case, there is one TEL structure for the circuit, and a second for its environment. This makes TEL structures both easier to compile to and easier to read. When this particular specification is broken up into processes, it is clear that the circuit itself is fairly simple, while the environment is more complex. TEL structures are defined formally in the next section, however, intuitively they can be thought of as a graphical representation of timed handshaking expansions. Each signal transition in a process corresponds to an *event* in the TEL structure. In the figure, events are shown as boxes connected by arrows. If the same transition occurs multiple times in a handshaking expansion it may occur multiple times in the TEL structure, however an optimizing step can often remove multiple occurrences of events. The arrows that connect events are referred to as *rules*, and are annotated with both a boolean expression and a lower and upper timing bound. Rules represent ordering relationships between events in a process. When two events occur sequentially in the handshaking expansion, a rule connects them. If there is a wait on a condition between these two events, the rule is annotated with

that wait, indicating that the second event cannot occur until both the first event has occurred and the condition has been satisfied. The timing bound, which distinguishes TEL structures from the previously described specification methods, allows the designer to specify a range on the delay between the firings of events in both the circuit and its environment.

The behavior of TEL structures can be illustrated by examining how the structure for the *sbuf-send-pk2* makes a choice based on the value of signal *done*. If *done* is low when sampled, the handshaking indicates that only the event *sendline-* can fire, otherwise events *sendline-* and *ack+* occur in parallel. This choice is represented in the TEL structure by the conflict relation (defined formally in the next section). If two events e_1 and e_2 conflict, indicated by $e_1 \# e_2$, either e_1 or e_2 can fire, but not both in the same iteration. In this circuit, there are two *sendline-* events, *sendline-/1* and *sendline-/2*, both of which cause the signal *sendline* to fall. However, the conflict relation states that only one of them can occur. Additionally, *ack+* conflicts with *sendline-/1*. Both the rules $sendline+ \rightarrow ack+$ and $sendline+ \rightarrow sendline-/2$ are annotated with the expression $\langle done \wedge ackline \rangle$, indicating that these rules cannot fire unless both *done* and *ackline* are high. This corresponds to the condition $done \wedge ackline$ in the handshaking expansion. The rule $sendline+ \rightarrow sendline-/1$ is annotated with the expression $\langle \neg done \wedge ackline \rangle$, corresponding to the other choice in the handshaking expansion. If *done* is low when *ackline* rises, event *sendline-/1* fires. If *done* is high when sampled, the TEL structure allows both *ack+* and *sendline-/2* to occur in parallel, just as specified in the handshaking expansion. This example shows how choices based on signal levels in handshaking expansions can be directly represented by TEL structures.

TEL structures can be used to represent specifications that are quite difficult to express with purely event based specification methods. Although they are no more expressive than general Petri nets, they are more expressive than the free choice Petri nets which are required by most STG synthesis methods. Since they allow processes to be separated, they significantly simplify compilation, increase readability, and make it possible to compile language constructs that involve levels. They also allow the designer to make timing assumptions in both the circuit and the environment which are not possible with the other specification methods.

2.2 The semantics of TEL structures

Event structures were introduced by Winskel [72] and timing has been added to them in several ways. Subrahmanyam added timing to event structures using temporal assertions [63]. Burns introduced timing in a deterministic version, the event-rule system, in which causality is represented using a set of rules, and a single delay value is associated with each rule [20]. Timed ER structures, introduced by Myers in [50], allow a delay range to be associated with each rule. TEL structures, described formally below, extend timed ER structures by allowing a boolean expression to be associated with each rule.

2.2.1 Timed event/level structures

TEL structures are based on timed ER structures, which are fundamentally acyclic. Cyclic specifications are represented by infinite timed ER structures, and state space exploration is done by dynamically creating the infinite unrolling of the specification until no new boolean states are possible. This type of acyclic semantics can also be used for TEL structures, but in order to make them more similar to the widely accepted specification methods such as Petri nets, TEL structures are defined here as cyclic structures.

Definition 2.2.1 *A TEL structure is a 6-tuple $T = \langle N, s_0, A, E, R, \# \rangle$ where:*

1. N is the set of signals;
2. $s_0 = \{0, 1\}^N$ is the initial state;
3. $A \subseteq N \times \{+, -\} \cup \$$ is the set of actions;
4. $E \subseteq A \times (\mathcal{N} = \{0, 1, 2, \dots\})$ is the set of events;
5. $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b : \{0, 1\}^N \rightarrow \{0, 1\})$ is the set of rules;
6. R_0 is the set of initially marked rules;
7. $\# \subseteq E \times E$ is the conflict relation.

The signal set, N , contains the wires in the circuit specification. The state s_0 contains the initial value of each signal in N . The action set, A , contains for each signal x in N , a rising transition, $x+$, and a falling transition, $x-$, along with the sequencing event $\$$, which is used to indicate an action that does not cause a signal transition. The event set, E , contains actions paired with instance indices (i.e., $\langle a, i \rangle$), which are used to distinguish multiple instances of a given signal transition within the specification. For example, there may be two possible situations in which a signal x can rise in a specification. These

rising actions on x are distinguished by having two events, $\langle x+, 1 \rangle$ and $\langle x+, 2 \rangle$. Pairing actions with instance indices allows an arbitrary number of events to be created from each action, including the sequencing action, $\$$. Sequencing events are often used to express nondeterminism where a signal may or may not transition. Although, formally the definition requires that all sequencing events be of the form $\langle \$, i \rangle$ where i is an integer, sequencing events of the form $\$s$ where s is a string are used in this thesis in order to make the purpose of the sequencing event more clear.

Rules represent causality between events. Each rule, r , is of the form $\langle e, f, l, u, b \rangle$ where:

1. e = enabling event,
2. f = enabled event,
3. $\langle l, u \rangle$ = bounded timing constraint, and
4. b = a boolean function over the signals in N .

A rule is *enabled* if its enabling event has occurred and its boolean function is true in the current state. There are two possible semantics concerning the enabling of a rule. In one semantics, referred to as *non-disabling semantics*, once a rule becomes enabled, it cannot lose its enabling due to a change in the state. In the other semantics, referred to as *disabling semantics*, a rule can become enabled and then lose its enabling. This can occur when another event fires, resulting in a state where the boolean function is no longer true. A single specification can include rules with both types of semantics. Non-disabling semantics are typically used to specify environment behavior and disabling semantics are typically used to specify logic gates. For the purposes of verification, the disabling of a boolean expression on a disabling rule is assumed to correspond to a failure, since it corresponds to a glitch on the input to a gate. A rule is *satisfied* if it has been at least l time units since it was enabled and *expired* if it has been at least u time units since it was enabled. Excluding conflicts, an event cannot occur until every rule enabling it is satisfied, and it must occur before every rule enabling it has expired.

The conflict relation, $\#$, is used to model disjunctive behavior and choice. When two events e and e' are in conflict (denoted $e \# e'$), this specifies that either e can occur or e' can occur, but not both. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. In the general case,

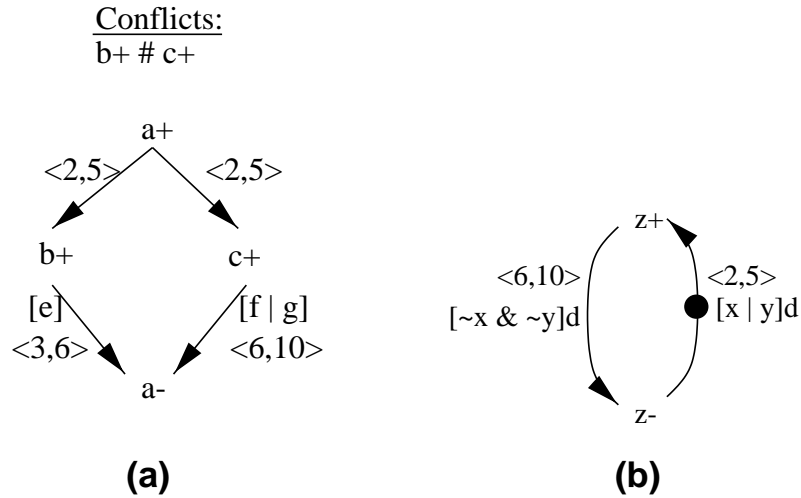


Figure 2.6. Examples of TEL structures.

an event is enabled when a maximal, non-conflicting set of its enabling events has fired. The ability for an event to fire when only a subset of its enabling events have fired models a form of disjunctive causality. Events that are enabled by multiple conflicting events are similar to merge places in Petri nets. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur. An event e that is the enabling event of multiple rules that have conflicting enabled events is similar to a choice place in a Petri net. Every pairwise conflict in the TEL structure must be specified, but this does not cause a problem for the user since TEL structures are typically generated from a higher level input language, such as VHDL [76].

2.2.2 Examples

Figure 2.6(a) shows an example of a TEL structure with non-disabling semantics. It has one conflict, $b+ \# c+$, which indicates that either the event $b+$ or the event $c+$ can occur after a firing of $a+$, but not both. The conflict also implies that only one of the signals $b+$ or $c+$ is necessary to fire $a-$. The rules $a+ \rightarrow b+$, and $a+ \rightarrow c+$ do not have level annotations. These rules function in exactly the same way as rules in standard ER structures and are enabled as soon as their enabling event, $a+$, fires. Since they have a bounded timing constraint of $\langle 2, 5 \rangle$, each of them becomes satisfied 2 time units after $a+$ fires and expired 5 time units after $a+$ fires. The rule $b+ \rightarrow a-$ has a level annotation, e , and does not become enabled until both $b+$ has fired and the signal e is true. It becomes satisfied 3 time units after it becomes enabled and expired 6 time units

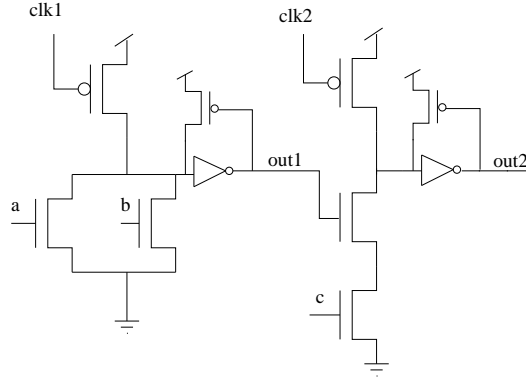


Figure 2.7. A delayed-reset domino gate.

after it becomes enabled. The rule $c+ \rightarrow a-$ also has a level annotation, $\langle f \vee g \rangle$, and becomes enabled after $c+$ has fired and $f \vee g$ is true. Since the semantics is non-disabling, once the expression has become true, the rule becomes satisfied after 6 time units, even if the expression later becomes false. Figure 2.6(b) shows an **or** gate represented as a TEL structure with disabling semantics, indicated by the “d” placed next to each level expression. The rule $z- \rightarrow z+$ becomes enabled when $z-$ has fired and $x \vee y$ is true. It becomes satisfied 2 time units later. If both x and y become false before $z+$ fires, the rule is disabled, and it is not satisfied again until 2 time units after $x \vee y$ becomes true again.

Figure 2.7 and 2.8 are used to illustrate how TEL structures are used to model circuits. Figure 2.7 shows a delayed-reset domino gate. The gate computes the function $(a \vee b) \wedge c$ in two stages. The first stage computes $a \vee b$ while $clk1$ is high, and the next stage computes $out1 \wedge c$ while $clk2$ is high. Both gates precharge while their respective clocks are low. Since neither n-stack has a “foot” transistor to ensure that the path to ground is turned off during the precharge phase, the timing of the circuit must guarantee that all the inputs to the gate are low by the time the local clock for each stage falls.

The TEL structure representation for the domino gate and its environment is shown in Figure 2.8. It includes one rising and one falling event for each signal. The specification indicates that there is a global clock $Gclk$ which rises 500 time units after it falls and falls 500 time units after it rises. The inputs to the gate, a , b , and c , nondeterministically rise some time after the clock rises. The nondeterminism is modeled using the conflict relation and sequencing events. Each rising event on an input conflicts with a corresponding

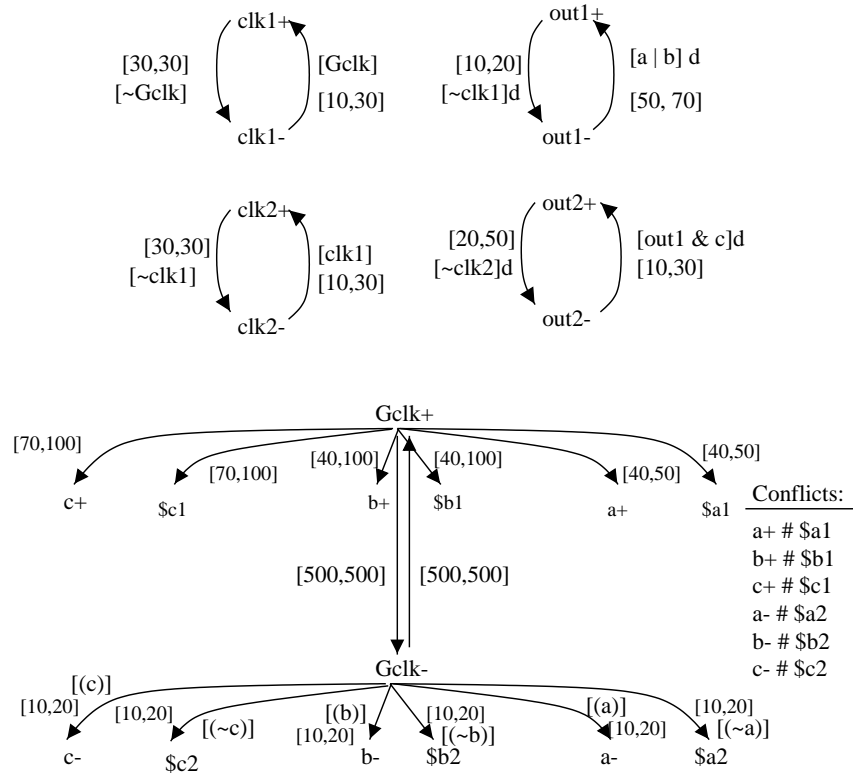


Figure 2.8. TEL structure for the gate in Fig. 2.7 and its environment.

sequencing event. Since the rising event and the sequencing event conflict, only one of them can occur. If the rising event for a signal fires, the signal rises in that clock cycle, if the sequencing event fires, it does not. A falling transition on the global clock is followed by falling transitions on all of the inputs, if they have risen. Again sequencing events and conflicts are used to deal with the nondeterminism. If an input signal rises on the rising edge of $Gclk$ then a falling event for that signal must occur when $Gclk$ falls. Otherwise, a conflicting sequencing event fires, preventing the falling event on the input signal from becoming enabled as soon as that signal rises again. The $Gclk$ signal also controls the firing time of the two local clocks, $clk1$ and $clk2$. The local clock $clk1$ rises between 10 and 30 time units after $Gclk$ rises and falls 30 time units after $Gclk$ falls. The other local clock, $clk2$ and the two gate outputs, $out1$ and $out2$ are specified in a similar fashion.

Although the TEL structure is readable for a small circuit, it would be difficult to specify a large macro at this level. ATACS provides support for two higher level input languages, VHDL and the timed handshaking expansions described earlier. Designers

```

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
entity level1 is
port(
    clk1 : in    std_logic;
    a : in    std_logic;
    b : in    std_logic;
    out1 : out  std_logic);
end level1;
architecture BEHAVIORAL of level1 is
signal out1_NEW : std_logic;
begin
out1 <= out1_NEW;
out1_NEW <= '1' after delay(50, 70) when a = '1' or b = '1' else
            '0' after delay(10, 30) when clk1 = '0' else
            out1_NEW;
end BEHAVIORAL;
entity level2 is
port(
    clk2 : in    std_logic;
    out1 : in    std_logic;
    c : in    std_logic;
    out2 : out  std_logic);
end level2;
architecture BEHAVIORAL of level2 is
signal out2_NEW : std_logic;
begin
out2 <= out2_NEW;
out2_NEW <= '1' after delay(10, 30) when c = '1'and out1 = '1' else
            '0' after delay(20, 50) when clk2 = '0' else
            req_NEW;
end BEHAVIORAL;

```

Figure 2.9. VHDL description of the domino gate

can specify circuits in these languages, and they are compiled into TEL structures using techniques described by Zheng and Myers in [76, 50]. Figure 2.9 shows the VHDL description of the two levels of logic used in the domino gate. The additional VHDL needed to connect up these gates is not shown here since it is simply structural VHDL to connect the signals. Since standard VHDL allows only deterministic delays, and timing verification requires bounded delays, the VHDL specification includes a package “`nondeterminism`” that allows for the use of the `delay` function. Calls to the delay function compile to a delay range in a TEL structure and are simulated in a VHDL simulator by making random delay choices within the delay range. Since these gates are state-holding, they cannot be specified using simple assignments in VHDL. Instead each signal has a rise condition and a fall condition, which can be compiled directly to rules in TEL structures. Using this interface, the designer can work in a familiar language without having to figure out how to represent circuits directly in TEL structures.

2.2.3 Timed Firing Sequences

The behavior specified by a TEL structure is defined with three types of operations: firing of rules, firing of events, and advancement of time. A time valued clock c_i is associated with each enabled rule r_i . A rule can fire when the clock meets the lower bound on the rule, and must fire when the clock reaches the upper bound on the rule. Using these semantics, the age of a clock never exceeds the upper bound of its associated rule. The firing of a rule may not immediately result in the firing of an event. An event fires when a *sufficient set* of the rules that enable it have fired. If all of the rules enabling an event e have non-conflicting enabling events, then e 's sufficient set is all of the rules that enable it. If some of the rules enabling e have conflicting enabling events then e has a number of different sufficient sets. For a set of rules, R_s , to be sufficient to fire e , all rules that enable e and are not in R_s must have enabling events that conflict with the enabling event of some rule in R_s . Events fire simultaneously with the last rule firing that creates a sufficient set of fired rules. Time is advanced using a function *max_advance*, which returns the maximum amount of time that can pass before a rule must fire or exceed its upper bound. These semantics define a set of firing sequences that contain both rule and event firings, where event firings are placed in the sequence immediately following the firing of its final enabling rule. In order for the analysis algorithm presented here to succeed in finding the state space of a TEL structure, it must be *one-safe*. In a one-safe

TEL structure, when the enabling event of a rule fires, it cannot fire again until either the enabled event of the rule fires, or an event that conflicts with its enabled event fires. This property is similar to the one-safe property on Petri nets, which prevents places from containing multiple tokens.

The set of behaviors of a TEL structure is defined by a set of sequences $\Sigma \in ((R^*)(E^*))^*$ where each firing (rule or event) is numbered sequentially. In order to simplify the notation, shorthand operations for dealing with firing sequences need to be defined. The function L is used to map an instance of a rule or event in the firing sequence back to the corresponding rule or event in the original specification, and the \in operator is used to specify whether a type of firing occurs in the sequence. Also, the functions l and u are used to return the lower and upper bound on a rule. Finally, it is useful to define a *choice_set* for each rule $r = \langle e, f, l, u, b \rangle$. The choice set of r contains all events which are enabled by e and conflict with f :

Definition 2.2.2 *The choice set of a rule $r = \langle e, f, l, u, b \rangle$ is defined as follows:*

$$choice_set(r) = \{f' \in E \mid \exists r' = \langle e, f', l', u', b' \rangle \in R \wedge f' \# f\}$$

When the event f fires, all of the events in the choice set of r require another firing of e before they have a chance to fire. Events that are not in the choice set of r do not require another firing of e in order to fire.

The state space of a TEL structure is found by exploring firing sequences of events and rules. The boolean state which is used to evaluate the boolean expressions associated with rules is defined by the rule firing sequence being explored, σ . The state resulting from a rule firing sequence, $\phi(\sigma)$ is simply the state that results when the firing sequence is executed starting from the initial state s_0 . We can now formally define what it means for a rule r to be enabled by a firing sequence σ .

Definition 2.2.3 *A rule $r = \langle e, f, l, u, b \rangle \in enabled(\sigma_{0..n})$ if one of the following conditions is true:*

1. $(r \in R_0) \wedge (\neg \exists \sigma_j \in \sigma_{0..n} : L(\sigma_j) = r) \wedge (\neg \exists \sigma_j \in \sigma_{0..n} : L(\sigma_j) \in choice_set(r)) \wedge (b(\phi(\sigma_{0..n})) \vee (non_disabling(r) \wedge \exists \sigma_j \in \sigma_{0..n} : b(\phi(\sigma_{0..j}))))$
2. $\exists \sigma_i \in \sigma_{0..n} : ((L(\sigma_i) = e) \wedge (\neg \exists \sigma_j \in \sigma_{i+1..n} : L(\sigma_j) = r) \wedge (\neg \exists \sigma_k \in \sigma_{i+1..n} : L(\sigma_k) \in choice_set(r)) \wedge (b(\phi(\sigma_{0..n})) \vee (non_disabling(r) \wedge \exists \sigma_l \in \sigma_{i+1..n} : b(\phi(\sigma_{0..l}))))))$

The first condition in the definition deals with rules that are initially marked. In order to satisfy the first condition, a rule must be initially marked (i.e. $r \in R_0$), and there must not be any other firing of the rule in the firing sequence (i.e. $\neg \exists \sigma_j \in \sigma_{0..n} : L(\sigma_j) = r$). There also must not be any other event firings in the sequence that would cause this rule to lose its chance to fire due to conflict (i.e. $\neg \exists \sigma_j \in \sigma_{0..n} : L(\sigma_j) \in \text{choice_set}(r)$). Finally, the boolean expression on the rule must either be satisfied by the current firing sequence, or be satisfied at some point in the current firing sequence for a non-disabling rule (i.e. $(b(\phi(\sigma_{0..n})) \vee (\text{non-disabling}(r) \wedge \exists : \sigma_j \in \sigma : b(\phi(\sigma_{0..j}))))$). This distinction is made since non-disabling rules only require that the boolean expression become true at some point before the rule fires. The second condition deals with all rule enablings other than the first firings of initially marked rules. In order for the second condition to hold, the firing sequence must contain a firing of the enabling event of the rule (i.e. $\exists \sigma_i \in \sigma_{0..n} : L(\sigma_i) = e$) and it must not contain a firing of the rule that occurs after the firing of the enabling event (i.e. $\neg \exists \sigma_j \in \sigma_{i+1..n} : L(\sigma_j) = r$). The firing sequence also must not contain a firing of an event in the choice set of r that occurs after the firing of e (i.e. $\neg \exists \sigma_k \in \sigma_{i+1..n} : L(\sigma_k) \in \text{choice_set}(r)$). Finally the boolean expression on the rule must either be satisfied by the current firing sequence or, if the rule is non-disabling, it must have been satisfied at some point in the sequence after the firing of the enabling event (i.e. $b(\phi(\sigma_{0..n})) \vee (\text{non-disabling}(r) \wedge \exists \sigma_l \in \sigma_{l+1..n} : b(\phi(\sigma_{0..l})))$).

When a sufficient set of rules has fired in the sequence, an event becomes enabled to fire. When an event fires, it “uses” the rule firings. Therefore, we need to define when a rule firing can be used to fire an event.

Definition 2.2.4 *The usable relation on a firing $\sigma_i : L(\sigma_i) = \langle e, f, l, u, b \rangle$ and firing sequence $\sigma_{0..n}$ is defined as follows:*

$$\text{usable}(\sigma_i, \sigma_{0..n}) \Leftrightarrow \neg \exists \sigma_j \in \sigma_{i+1..n} : (L(\sigma_j) = f) \vee (L(\sigma_j) \in \text{choice_set}(L(\sigma_i))).$$

This definition means that a rule firing is usable until its enabled event fires or an event in its choice set fires. A rule $r = \langle e, f, l, u, b \rangle$ remains usable when an event, f' , that conflicts with f fires, if f' and f do not share e as an enabling event. For example, consider the TEL structure in Figure 2.10, and assume that $a+$ and $d+$ have fired. A firing of $a+ \rightarrow c+$ is made unusable by the firing of event $b+$ since $b+$ is in the choice set of $a+ \rightarrow c+$. However, the firing of $b+$ does not make a firing of $d+ \rightarrow c+$ unusable. This distinction is made since another firing of $a+$ is necessary before $c+$ can fire, but

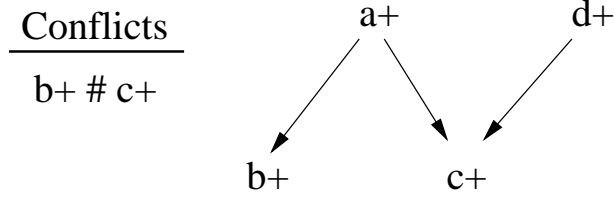


Figure 2.10. Conflict behavior.

another firing of $d+$ is not necessary before $c+$ can fire.

Events fire when there is a sufficient set of usable rules.

Definition 2.2.5 *The set of firable events of a firing sequence $\sigma_{0..n}$ is defined as follows:*

$$\text{firable}(\sigma) = \{f \in E \mid \forall r = \langle e, f, l, u, b \rangle \in R : \exists \sigma_i \in \sigma_{0..n} : L(\sigma_i) = r \wedge \text{usable}(\sigma_i, \sigma_{0..n}) \vee \\ \exists \sigma_j \in \sigma_{0..n} : L(\sigma_j) = \langle e', f, l, u, b \rangle \wedge e' \# e \wedge \text{usable}(\sigma_j, \sigma_{0..n})\}$$

The firable set contains all events which have a sufficient set of usable rules in the firing sequence. All of the rules that enable an event must either have a usable firing in σ or have an enabling event which conflicts with a rule that has a usable firing in σ .

Definition 2.2.5 allows us to define the set of sequences which are allowed by the TEL structure, $\Sigma \in ((R^*)(E^*))^*$ as follows:

Definition 2.2.6 *A sequence $\sigma \in \Sigma$ if and only if $\forall \sigma_i$:*

1. $L(\sigma_i) \in R \Rightarrow L(\sigma_i) \in \text{enabled}(\sigma_{0..i-1})$
2. $L(\sigma_i) \in E \Rightarrow L(\sigma_i) \in \text{firable}(\sigma_{0..i-1})$
3. $L(\sigma_i) \in R \wedge \text{firable}(\sigma_{0..i}) \neq \emptyset \Rightarrow \sigma_{i+1} \in \text{firable}(\sigma_{0..i})$

The first requirement states that rules must be enabled when they fire. The second requirement of this definition states that all events must be in the firable set when they fire. The third requirement is that if the firable set of a rule firing is not empty, an event in the firable set must follow it in the sequence.

Each rule firing σ_i can be associated with the event firing that enabled the rule by the causal event function, E_c , defined as follows :

Definition 2.2.7 $E_c(\sigma_i, \sigma) = \sigma_j$ where j is the maximum value less than i for which $L(\sigma_i) \notin \text{enabled}(\sigma_{0\dots j-1})$ and $L(\sigma_i) \in \text{enabled}(\sigma_{0\dots j})$

This means that the causal event for a rule firing is the event firing which causes the rule to become enabled. This event may either be the enabling event for the rule or it may be an event that changes the value of a signal that causes the boolean expression associated with the rule to evaluate to true.

Any sequence can be given a *timing assignment* τ which maps an event to the time at which it occurs. For each sequence, $\sigma \in \Sigma$, there is a set of *valid* timing assignments, referred to as $\text{valid}(\sigma)$.

Definition 2.2.8 A timing assignment τ is valid for a sequence σ if :

$$\begin{aligned} \forall \sigma_i \in \sigma : \tau(\sigma_i) \leq \tau(\sigma_{i+1}) \wedge \\ L(\sigma_i) \in E \Rightarrow \tau(\sigma_i) = \tau(\sigma_{i-1}) \wedge \\ L(\sigma_i) \in R \Rightarrow \tau(E_c(\sigma_i, \sigma)) + l(L(\sigma_i)) \leq \tau(\sigma_i) \leq \tau(E_c(\sigma_i, \sigma)) + u(L(\sigma_i)). \end{aligned}$$

This means that a timing assignment is valid if it corresponds to the order of the firing sequence, all events fire simultaneously with the rule immediately preceding their firing, and rules fire between their lower and upper bounds after their causal event. A firing sequence $\sigma \in \Sigma$ is reachable in the specification TEL structure if and only if it can be given a valid timing assignment.

2.2.4 Examples of Firing Sequences

In order to clarify the formalism presented in the previous section, this section presents some examples of timed firing sequences which are reachable in the specification shown in Figures 2.3 and 2.4. Any valid sequence must begin with some interleaving of rule firings $[\text{ackline-}/2, \text{req+}]$, $[\text{req-}, \text{req+}]$, and $[\text{done-}, \text{req+}]$ since these are the only rules that are initially enabled. Any sequence that does not begin with these rules it not reachable. Suppose that the firing sequence begins with:

$[\text{ackline-}/2, \text{req+}]$, $[\text{done-}, \text{req+}]$, $[\text{req-}, \text{req+}]$, req+

The rule firings can be given any timing assignment between two and five, as long as they are monotonically increasing. For example:

$\{[\text{ackline-}/2, \text{req+}], 2\}$ $\{[\text{done-}, \text{req+}], 3\}$ $\{[\text{req-}, \text{req+}], 4\}$ $\{\text{req+}, 4\}$

is a valid timed firing sequence. This sequence:

$\{[\text{ackline-}/2, \text{req+}], 3\}$ $\{[\text{done-}, \text{req+}], 2\}$ $\{[\text{req-}, \text{req+}], 4\}$ $\{\text{req+}, 4\}$

is not a valid timed firing sequence since the firing times are not monotonically increasing.

This sequence:

{[ackline-/2, req+],2} {[done-, req+],3} {[req-, req+],4} {req+,5}

is also invalid since the event firing $req+$ does not fire simultaneously with the last rule in its sufficient set firing. Also note that after $req+$ fires there are no usable rule firings.

The only rule firing that is enabled by the valid firing sequence:

{[ackline-/2, req+],2} {[done-, req+],3} {[req-, req+],4} {req+,4}

is $[ack-, sendline+]$. When this rule and the event $sendline+$ fire, the following firing sequence is produced:

[ackline-/2, req+], [done-, req+], [req-, req+], req+, [ack-,sendline+],
sendline+

In this sequence, $req+$ is causal to $[ack-, sendline+]$ since $[ack-, sendline+]$ is not enabled before $req+$ fires, and is enabled after $req+$ fires. Since $[ack-, sendline+]$ is the causal rule for $sendline+$, $req+$ is causal to $sendline+$. Therefore, a valid timing assignment must allow between two and five time units between $req+$ and $sendline+$.

The following is a valid timing assignment:

{[ackline-/2, req+],2} {[done-, req+],3} {[req-, req+],4} {req+,4},
{[ack-, sendline+],7}, {sendline+,7}

The firings, $[req+, ackline+]$, $ackline + /1$ are added to the firing sequence in a similar manner resulting in the following timed firing sequence:

{[ackline-/2, req+],2} {[done-, req+],3} {[req-, req+],4} {req+,4},
{[ack-, sendline+],7}, {sendline+,7}, {[req+, ackline+/1],14},
{ackline+/1,14}

Although there are now multiple rules whose enabling events have fired, only one rule, $[sendline+, sendline - /1]$, has a boolean expression which is satisfied by the state. This rule, and the event $sendline - /1$ are now added to the sequence:

{[ackline-/2, req+],2} {[done-, req+],3} {[req-, req+],4} {req+,4},
{[ack-, sendline+],7}, {sendline+,7}, {[req+, ackline+/1],14},
{ackline+/1,14}, {[sendline+, sendline-/1], 18}, {sendline-/1,18}

The event $ackline + /1$ is causal to $[sendline+, sendline - /1]$ so a timing assignment four time units after $ackline + /1$ fires is valid for $[sendline+, sendline - /1]$ and $sendline - /1$.

There are now three enabled rules, $[ackline+/1, ackline-/3]$, $[ackline+/1, ackline-/1]$, and $[ackline+/1, done+]$. Each of these rules has the others in its choice set, so when one of them fires, other rules lose their enabling. Suppose that $[ackline+/1, done]$ is chosen to fire. The new timed firing sequence is as follows:

```
{[ackline-/2, req+],2} {[done-, req+],3} {[req-, req+],4} {req+,4},
{[ack-, sendline+],7}, {sendline+,7}, {[req+, ackline+/1],14},
{ackline+/1,14}, {[sendline+, sendline-/1], 18}, {sendline-/1,18},
{[ackline+/1, done+], 20}, {done+, 20}
```

The rule $[ackline+/1, done+]$ has a conflict set consisting of $[ackline+/1, ackline-/3]$, since they share the same enabling event and have conflicting enabled events. Since $[ackline+/1, done+]$ has fired in the sequence, $[ackline+/1, ackline-/3]$ loses its enabling and does not get another chance to fire until the event $ackline+/1$ fires again.

This firing sequence can, of course continue on indefinitely. However, the short subsequence above illustrates most of the concepts about firing sequences defined in this chapter. Since sequences are infinite, it is necessary to define equivalence classes between sequences, so that an algorithm can determine when it is in a state it has seen before and can stop adding new firings to the sequence. This is the topic of the next chapter.

2.3 Summary

This chapter defines the TEL structure formalism and the set of timed firing sequences that are specified by a TEL structure. TEL structures conform more closely to circuit behavior than purely event based formalisms. This makes it is easier to construct circuit specifications using TEL structures than it is to construct them using purely event based specification methods.

CHAPTER 3

GEOMETRIC TIMING ANALYSIS

Time is what prevents everything from happening at once.

- John Archibald Wheeler

In order to do synthesis or verification of a TEL structure specification, it is necessary to find all of its allowable boolean states. The number of boolean states is finite, but the number of firing sequences generated by a cyclic specification is infinite. An algorithm that tries to find all boolean states by exploring all firing sequences never completes. Therefore, it is necessary to define *equivalence classes* between firing sequences. Any two firing sequences that are in the same equivalence class are guaranteed to lead to the same future set of boolean states. Equivalence classes allow an algorithm to know when to stop. When it finds a sequence which is in the same equivalence class as a previously explored sequence, the algorithm knows that the current sequence cannot result in any new behavior. This prevents the algorithm from attempting to explore infinitely long sequences which allows it to complete. This chapter defines equivalence classes for timed firing sequences and presents an algorithm for timed state space exploration based on these equivalence classes.

3.1 Defining Equivalence Classes

The infinite nature of the set of timed firing sequences is two-fold. The number of sequences $\sigma \in \Sigma$ that have valid timing assignments is infinite. Additionally, each individual sequence can have an infinite number of valid timing assignments. State space exploration requires that this infinite set of sequence, timing assignment pairs be divided into a finite set of equivalence classes. The obvious way to do this in the untimed case is to say that two sequences σ and σ' represent equivalent states if the set of enabled rules that results from executing σ and σ' is the same. Therefore, for state space exploration, the *untimed state* of the system is simply the set of enabled rules. This is equivalent to

two Petri net firing sequences being in the same equivalence class if they have the same marking. The timed state of the system is represented by a set of *active clocks*. An active clock is created whenever a rule becomes enabled, and eliminated when the rule fires. After a firing sequence is executed, there is an active clock for every rule that is enabled by the execution of the firing sequence. The set of possible timing assignments to the sequence determines the set of possible ages that the active clocks can have. This set of ages represents the timed state of the specification at the end of the firing sequence. Therefore, two firing sequences can be said to lead to the same timed state if they result in the same set of enabled rules, and the sets of possible ages for the active clocks resulting from the two sequences are the same.

In order to determine the age of an active clock, c_i , it is necessary to know which event firing enabled the rule, r_i , associated with it. Each enabled rule, r_i , can be associated with the event firing that enabled it by a modification of the causal event function, E_c , defined in the previous chapter. The new function, E_m (for marking event), is defined as follows:

Definition 3.1.1 $E_m(r_i, \sigma_{0..n}) = \sigma_j$ where j is the maximum value for which $r_i \notin \text{enabled}(\sigma_{0..j-1})$ and $r_i \in \text{enabled}(\sigma_{0..j})$

This means that the marking event for an enabled rule is the event firing which causes the rule to become enabled. This event may either be the enabling event for the rule or it may change the value of a signal that causes the boolean expression associated with the rule to evaluate to true.

Definition 3.1.1 can be used to formally define *max_advance*, the function that determines how much time can advance without forcing a rule to fire for a firing sequence σ of length n .

Definition 3.1.2 The function *max_advance*($\sigma_{0..n}, \tau$) is defined as follows:

$$\text{max_advance}(\sigma_{0..n}, \tau) = \min_{r_i \in \text{enabled}(\sigma_{0..n})} (u(r_i) - (\tau(\sigma_n) - \tau(E_m(r_i, \sigma_{0..n}))))$$

The maximum amount that time can advance without a rule r_i exceeding its upper bound is $(u(r_i) - (\tau(\sigma_n) - \tau(E_m(r_i, \sigma_{0..n}))))$, which is the difference between the upper bound on r_i and its current age. The definition of *max_advance* returns the minimum of this expression over all enabled rules. This is the maximum amount of time that can pass before at least one rule must fire or exceed its upper bound.

The *max_advance* function is used to determine all of the possible clock ages that are allowed by a timing assignment, τ , for a sequence of length n . It is computed as follows:

Definition 3.1.3 If $r_i \in \text{enabled}(\sigma_{0..n})$, c_i must satisfy the following inequality:

$$\tau(\sigma_n) - \tau(E_m(r_i, \sigma_{0..n})) \leq c_i \leq (\tau(\sigma_n) - \tau(E_m(r_i, \sigma_{0..n})) + \text{max_advance}(\sigma_{0..n}, \tau))$$

This means that a clock is no younger than the time difference between the firing of the event that created it and the last event to fire in the sequence, and it must not exceed an age that would force another rule to fire. The set of values for a clock c_i that are allowed by a timing assignment τ are referred to as $\tau(c_i)$. Since the ages of the clocks determine which future states are possible, two sequences σ and σ' can be said to have the same *timed state* if $\text{enabled}(\sigma) = \text{enabled}(\sigma')$ and τ is a valid timing assignment to σ if and only if there is a valid timing assignment τ' to σ' such that $\forall r_i \in \text{enabled}(\sigma) : \tau(c_i) = \tau'(c_i)$. This definition means that if the clock ages that can result from firing σ and σ' are the same, the two sequences result in the same futures, and are therefore considered equivalent.

3.2 Timed State Space Exploration

Suppose that there exists a representation M which gives the ages of the clocks allowed by a firing sequence. A timed state, TS , then consists of $\text{enabled}(\sigma) \times M$. This representation of a timed state allows two sequences to be compared to see if they have the same timed state, and an algorithm which explicitly examines firings sequences could be developed to explore the state space. However, firing sequences can be very long. Storing and manipulating them would take a large amount of memory and time. In order to produce acceptable performance, a state space exploration algorithm must compactly store the useful information from the firing sequence without storing the entire firing sequence.

In order to develop an algorithm to find all of the timed states without saving sequences of firings, more information needs to be stored in the timed state. The boolean state resulting from the sequence is necessary in order to compute a set of enabled rules. Since the algorithm does not store the sequence itself, it must store the current state, and update it whenever an event fires. Therefore, the current state, s_c , is added to the timed state for use in the algorithm. Also, if the sequence is available, it is simple to compute the set of rules whose enabling events have fired but whose enabled events

have not. This set is not the same as the set of enabled rules, since it does not consider boolean expressions. However, this set is necessary in order to compute the set of enabled rules since only these rules are eligible to become enabled. This set is referred to as R_m (for marked set) in the algorithm and added to the timed state. Next, since the algorithm is not storing the sequence, it cannot compute the set of enabled rules from the sequence. It must store this set as well by adding and removing rules as events fire. In the algorithm the set R_{en} replaces $enabled(\sigma)$ in the timed state. It may seem that adding both R_m and R_{en} to the timed state is redundant, however it is necessary since rules can be disabling or non-disabling. When non-disabling rules become enabled, they remain enabled regardless of the current boolean state. Therefore, it cannot be determined whether non-disabling rules are enabled simply from looking at R_m and s_c . Disabling rules can lose and regain their enabling many times before they fire depending on the current boolean state. Therefore it is necessary to record which rules are currently marked so that the algorithm knows which rules to check and possibly add to R_{en} . Finally, if the algorithm were operating on a sequence, it could determine the set of usable rule firings from the sequence. Since it cannot look at the whole sequence, it must maintain another set, R_f (for fired rules), which contains usable rule firings. This set is necessary in order for the algorithm to determine which events can fire. With these additions, the timed state for use in the algorithm now is as follows: $R_m \times R_{en} \times s_c \times R_f \times M$.

Using this representation, the timed state space of a TEL structure can be explored using the algorithm in Figure 3.1. The algorithm does a depth-first search of the timed state space, finding all the timed states that are reachable. It first initializes all of the elements of the timed state. The set R_m , is set to R_0 , the set of initially marked rules in the TEL structure. The current state is set to the initial state of the TEL structure. The R_{en} set is created by including all marked rules whose boolean expressions are satisfied by the initial state. The timing information, M , is then initialized for all the enabled rules. All initially enabled rules have a minimum age of zero and a maximum age of the least upper bound among them. Their relative age differences are all set to zero. The algorithm then initializes R_f to \emptyset . After these steps, the algorithm has created the initial timed state. It combines all the elements of the timed state into a data structure, TS , and adds it to the state space Φ . In order to use the state space for synthesis, the algorithm also must store the set of possible transitions between states. This set is called Γ , and is initially empty. After initializing Γ , the algorithm calls the function *find_timed_enabled*

Algorithm 3.2.1 (Find timed states)

```

state_space find_timed_states(TEL structure TEL = ⟨N, s0, A, E, R, R0, #⟩){
  Rm = R0;
  sc = s0;
  Ren = {⟨e, f, l, u, b⟩ ∈ Rm : b(sc)};
  M = initialize_timing(Ren, TEL);
  Rf = ∅
  timed_state TS = Ren × Rm × sc × Rf × M;
  set_of_states Φ = {TS};
  set_of_transitions Γ = ∅;
  rule_list RL = find_timed_enabled(TS, TEL);
  bool done=false;
  while (¬done){
    event_fired = false;
    rule r = ⟨e, f, l, u, b⟩ = head(RL);
    push(TS, tail(RL));
    Rold = Ren;
    Rf = Rf ∪ r;
    Rm = Rm - r;
    Ren = Ren - r;
    if (∀ ri = ⟨ei, f, li, ui, bii ∈ Rf) ∨ (∃ rj = ⟨ej, f, lj, uj, bif : ei#ej))) {
      event_fired = true;
      if (f = ⟨xi+, m⟩) sc[s_index(xi)] = 1;
      else if (f = ⟨xi-, m⟩) sc[s_index(xi)] = 0;
      Rm = Rm - {rj ∈ R : f ∈ choice_set(rj)};
      Rm = Rm ∪ {⟨ei, fi, li, ui, bii = f};
      Rf = Rf - {⟨e, fi, li, ui, bif : fi = f};
      Rf = Rf - {rj ∈ R : f ∈ choice_set(rj)};
      Ren = Ren - {rj ∈ R : f ∈ choice_set(rj)};
      Ren = Ren ∪ {rj ∈ Rm : bi(sc)};
      foreach (ri = ⟨ei, fi, li, ui, bien ∪ Rf : ri is disabling)
        if (¬bi(sc))
          if (fail_on_disable) return fail;
          else Ren = Ren - ri;
    }
    M = update(TEL, M, r, Ren, Ren - Rold, event_fired);
    TSold = TS;
    TS = Ren × Rm × sc × Rf × M;
    if (TS ∉ Φ) then
      Φ = S ∪ {TS};
      Γ = Γ ∪ {(TSold, TS)};
      RL=find_timed_enabled(TS, TEL);
    else if (TS ∈ Φ) then
      Γ = Γ ∪ {(TSold, TS)};
      if (stack is not empty) then (TS, AL)=pop();
      else done = true;
  }
  return (Φ, Γ);
}

```

Figure 3.1. Timed state space exploration.

Algorithm 3.2.2 (Find timed enabled)

```

rule_list RL find_timed_enabled( $TS \langle R_{en}, R_m, s_c, R_f, M \rangle, TEL \langle N, s_0, A, E, R, R_0, \# \rangle$ ){
  rule_list RL =  $\emptyset$ ;
  for each ( $r = \langle e, f, l, u, b \rangle \in R_{en}$ ){
    if ( $min\_clock\_value(M, r) \geq l$ ) add_list(RL, r);
  }
  return RL;
}

```

Figure 3.2. Find timed enabled rules.

which returns the set of rules that are currently allowed to fire. The function is defined in Figure 3.2. It goes through all of the enabled rules and adds those whose clocks meet their lower bounds to the list of rules that can fire. The method for extracting the minimum age of a rule's clock from the representation of the timing is discussed in the next section. The algorithm has now initialized everything and is ready to begin the main loop.

The main loop of the algorithm continues until all of the reachable states have been found, a condition represented by the variable *done*. When the loop begins, the function removes the rule it is going to fire, r , from the front of the rule list (i.e. $head(RL)$) and places the rest of the rule list ($tail(AL)$), and the timed state on the stack. Next, it saved the current R_{en} set by assigning it to R_{old} . This is done so that the algorithm can determine which rules in R_{en} are newly added. It then adds r to the fired set since it is firing, and removes it from R_m and R_{en} since it is no longer available to fire. Next, the algorithm checks if firing of r causes an event to fire. An event fires if all of the rules that enable it are either in R_f or have enabling events that conflict with the enabling event of a rule that is in R_f . If an event can fire, the algorithm updates the state vector, using the *s_index* function to find the index of the signal that is changing state in the state vector. If a sequencing event fires, the state vector remains unchanged. Next the algorithm updates the rule sets to reflect the firing of a new event. The marked set, R_m , loses all rules that contain f in their choice sets, since they have lost their chance to fire. The marked set gains all rules that have f , the firing event, as their enabling event. The fired set loses all rules that enable f , and all rules that contain f in their choice sets. These rules are no longer usable since they have either been used or become unusable due the the firing of an event in their choice sets. The enabled set is also updated: it loses all rules that contain f in their choice sets and gains all rules in R_m whose boolean expressions are satisfied in the new state. The algorithm then checks for rules that have

been disabled. If a disabling rule is in the enabled set and its boolean expression is no longer true due to the firing of f , it has been disabled. This can result in two different outcomes. If the designer wishes to consider disabling failures, since they correspond to hazards on the inputs of gates, then at this point the algorithm returns a fail condition. If the designer does not want the algorithm to fail on a disabling, the offending rule is removed from the enabled set and the algorithm continues. After all the rule sets have been updated, the algorithm updates the timing information M . The details of this are discussed in the next section. Next, the old timed state is saved in TS_{old} and all of the sets are combined into the new timed state. The algorithm then checks to see if this new state is already in the state space. If it is not in the state space, the new state is added to Φ and a new transition, from TS_{old} to TS , is added to the transition set Γ . Then a new list of rules to fire is computed from the current state. If the current state is already in Φ , the algorithm removes a state and rule list from the stack and continues the main loop. If the stack is empty, then there are no more new states to be found and the algorithm is completed.

Untimed states are only explored if they can be reached given the timing information in the specification. This can eliminate large portions of the untimed state space for some designs when the algorithm is used for synthesis. Many states that are reachable without timing information are not reachable given the timing constraints in the specification. However, the algorithm explores the entire *timed* state space, and the size of the timed state space depends on the representation chosen for the timing information.

3.3 Representing Time

The timing analysis algorithm presented here uses geometric regions (also known as zones) to represent the timing information within a timed state. As discussed earlier in the chapter, whenever a rule r_i is enabled, a clock c_i is created to be used in timing analysis. The minimum and maximum age differences of all the clocks are stored in a *constraint matrix* M (also known as a difference bound matrix). Each entry m_{ij} in the matrix M has the value $\max(c_j - c_i)$, which is the maximum age difference of the clocks. A dummy clock c_0 whose age is always 0 is also included. The maximum age difference between c_i and c_0 (m_{0i}) is the maximum age of c_i . The maximum age difference between c_0 and c_i (m_{i0}) is the negation of the minimum age of c_i . Note that M only needs to contain information on the timing of currently enabled rules, not on every rule in the

TEL structure. This particular way of representing timed regions was first introduced by Dill in [28]. The constraint matrix represents a convex $|R_{en}|$ dimensional region. Each dimension corresponds to an enabled rule, and the age at which a rule fires can be anywhere within the space.

Many matrices can be used to represent the same region in space since some entries may be underconstrained. However, there is a canonical representation where every constraint is *maximally constraining*. A set of constraints is maximally constraining if each constraint can reach its maximum value for some timing assignment without violating any of the other constraints. In the algorithm, the matrix is made maximally constraining through a process called *recanonicalization*. Recanonicalization takes a matrix M where some of the m_{ij} 's are greater than $\max(c_j - c_i)$ and produces a matrix where all the m_{ij} 's have their maximum allowed value. The assignment of the m_{ij} 's so that they all have their maximum value is always unique, so the algorithm can determine when a given region is equivalent to or contained in a region that has been seen before. Recanonicalization is essentially the all pairs shortest path problem and can be done in $O(n^3)$ time with Floyd's algorithm [28].

Geometric regions are used by Rokicki in `Orbits` [58, 59, 53] to do timed state space exploration on Orbital net specifications with the single behavioral place restriction. The single behavioral place restriction is made in `Orbits` to ensure that the geometric regions that represent the time behavior of the system are always convex. If the values of clocks can exceed their upper bounds, the regions representing the time behavior may not be convex. Figure 3.3 shows an example of this. In this specification, either the separation between $a+$ and $c+$ must not exceed 5, or the separation between $b+$ and $c+$ must not exceed 4. Since TEL structure semantics does not require both upper bound constraints to be met, the resulting region is non-convex. Since Floyd's algorithm only works on convex regions, this must be avoided. However, when rules are allowed to fire independently of events as they are in the state space exploration algorithm section, clocks can no longer exceed their upper bounds, and the regions are guaranteed to be convex. In this example, 2 regions would be generated to cover the space shown in the figure.

The algorithm in Figure 3.4 shows how the function for updating timing information used in Figure 3.1 is implemented with geometric regions. The function takes as input the TEL structure specification, the constraint matrix, the rule that is firing, the set of enabled rules, the set of newly enabled rules, and a bit which indicates if an event is

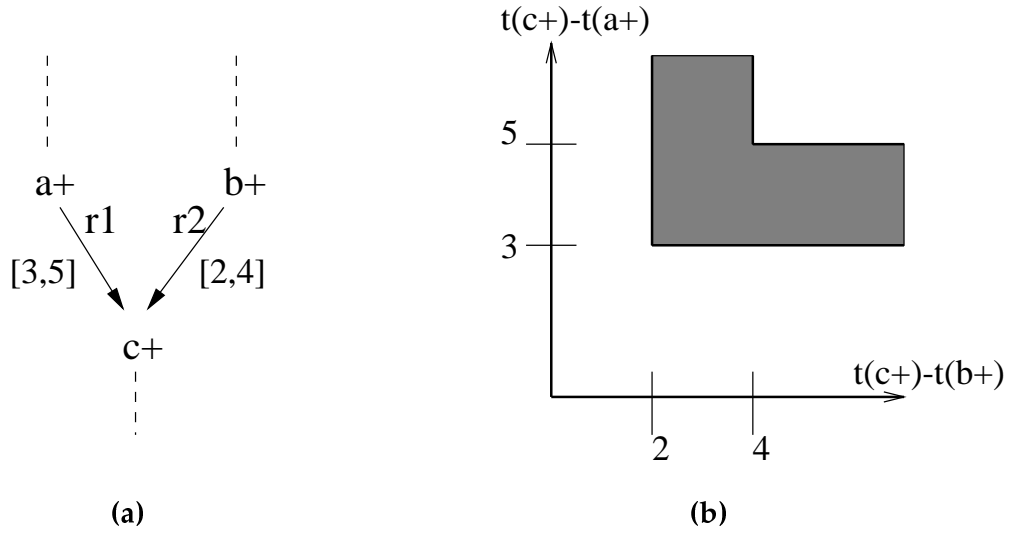


Figure 3.3. (a) TEL structure with multiple behavior rules, and (b) non-convex region.

Algorithm 3.3.1 (Update)

```

void update(TEL structure TEL  $\langle N, s_0, A, E, R, R_0, \# \rangle$ , geometric region  $M$ ,
           rule  $r = \langle e, f, l, u, b \rangle$ , rule set  $R_{en}, R_{new}$ , bool event_fired) {
  if ( $M[index(r)][0] > -l$ ) then  $M[index(r)][0] = -l$ ;
  recanonicalize( $M$ );
  project( $M$ ,  $index(r)$ );
  if(event_fired){
    forall( $r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{new}$ ) {
       $M[0][index(r_i)] = 0$ ;
       $M[index(r_i)][0] = 0$ ;
      forall( $r_j \in R_{new}$ )
         $M[index(r_j)][index(r_i)] = 0$ ;
         $M[index(r_i)][index(r_j)] = 0$ ;
      forall( $r_j \in R_{en} - R_{new}$ )
         $M[index(r_j)][index(r_i)] = M[index(r_j)][0]$ ;
         $M[index(r_i)][index(r_j)] = M[0][index(r_j)]$ ;
    }
  }
  forall( $r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en}$ )
     $M[0][index(r_i)] = u_i$ ;
  recanonicalize( $M$ );
  normalize( $M$ );
}

```

Figure 3.4. Procedure for updating the geometric region.

firing. The *index* function used in the algorithm takes a rule, and returns the index in the constraint matrix that corresponds to it. The first step of the function is to check if the minimum age of the firing rule's clock in the matrix is less than the lower bound on the rule. If it is, the lower bound on the age of the rule in the matrix is set to the negation of minimum age of the rule. This ensures that the minimum age of each clock is no less than the difference between the time it is created and the time that the last event in the sequence fires. The row and column corresponding to the fired rule is then removed from the matrix by the *project* operation. Next, if an event fired, the algorithm adds clocks for newly enabled rules. A rule is newly enabled if it is in R_{new} . When a rule is initially enabled, its age is zero, so the entries in the matrix for its minimum and maximum age are set to zero. If a set of rules is enabled at the same time, the relative ages difference between all pairs of rules in the set is zero. Therefore, entries in the matrix representing age differences between events in R_{new} are set to zero. Age relationships between the new rules and the previously existing ones must also be entered in the matrix. The maximum age difference between a new rule and any previously existing rule is just the maximum age of the previously existing rule. Therefore, the new maximum age difference entries are copied from row zero of the matrix which contains the maximum ages of existing rules. The minimum age difference between the new rule and a previously existing rule is the minimum age of the previously existing rule, and this minimum age is copied from column zero of the matrix. Finally, the algorithm sets the maximum age of each rule in the matrix to its specified maximum age, u , and recanonicalizes the matrix. This allows time to advance as far as possible without causing any rule to exceed its maximum age. The final step is normalization. The normalization step is necessary to deal with rules that have infinite upper bounds and is described in detail by Rokicki in [58]. The new region now represents all possible clock ages given the firing sequence that is currently being explored.

3.4 Examples

Figure 3.5 shows an example of how the geometric algorithm would be applied to the simple TEL structure shown at the top of the figure. The first column shows the constraint matrix at each step and the second column shows the region in space represented by the matrix. The recanonicalization procedure that is applied after each step is not shown here, but is described in detail by Rokicki in [58]. Initially, rules r_1 and r_2 which have

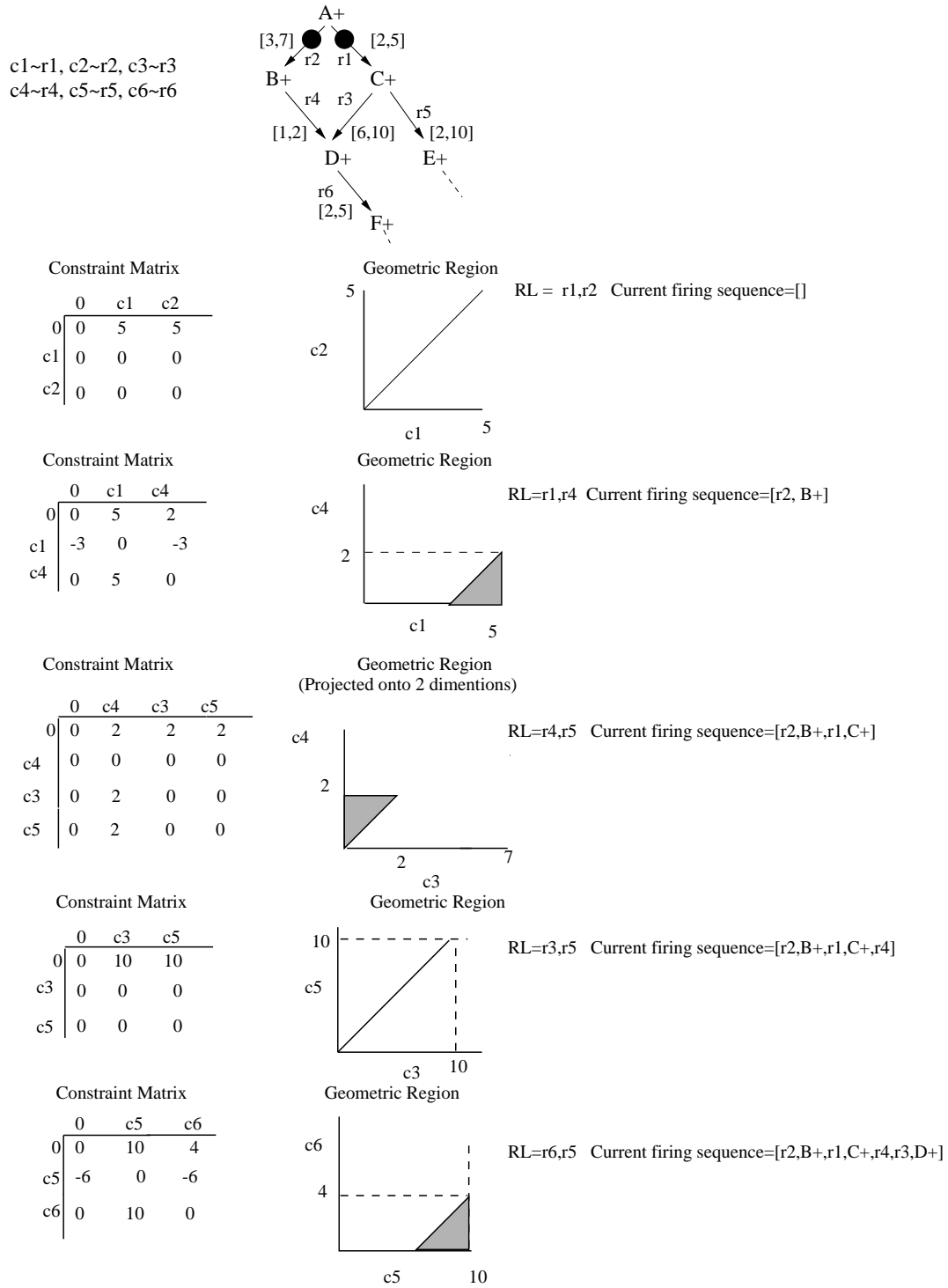
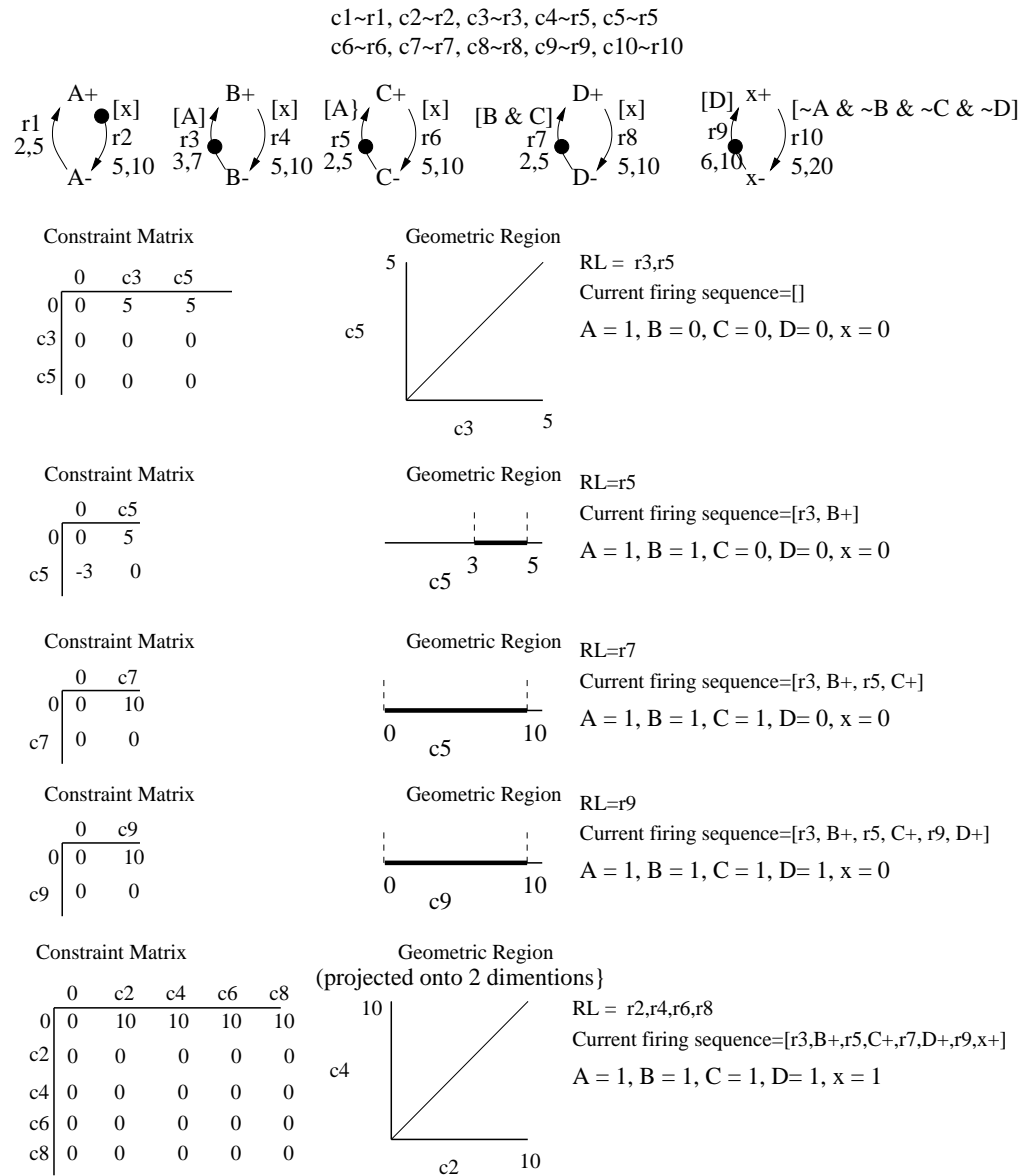


Figure 3.5. Firing rules.

tokens to indicate they are initially marked, are given clocks c_1 and c_2 . The initial constraint matrix indicates that the maximum age for both clocks is 5. Since the lower timing bounds on both r_1 and r_2 are less than 5, they are both added to the list of rules that can fire, RL . The rule r_2 is chosen to fire. The clock for r_2 is projected out of the constraint matrix, and the matrix is constrained so that all clocks that existed when r_2 fired must have a minimum age of 3. A new clock is added for the newly enabled rule, r_4 . It must be between 3 and 5 time units younger than the clock for r_1 since the clock for r_1 has an age between 3 and 5 time units when it is added. The list of rules to fire now contains r_1 and r_4 . The rule r_1 is chosen to fire next, causing rules r_3 and r_5 to become enabled. The new rule list contains r_4 and r_5 but not r_3 since the lower bound on r_3 is 6, and the maximum age for r_3 allowed by the matrix is 2. Next, r_4 is chosen to fire. It does not cause an event to fire, so no new clocks are added to the constraint matrix. After r_4 fires, the maximum age of the rule r_3 can advance to 10, allowing it to be placed on the new rule firing list. The rule r_3 can then fire, producing the last matrix and region in the figure.

The next example, shown in Figure 3.6, demonstrates how the algorithm works with level expressions. The TEL structure at the top of the figure has similar behavior to the TEL structure shown in Figure 3.5, but it specifies the behavior using levels instead of events. In the initial state, all of the signals are set to zero, except for A which is set to one. Rules r_3 and r_5 are enabled and have entries in the constraint matrix. They are also both on the rule list since their lower bounds are reached by the maximums in the matrix. Rule r_3 is chosen to fire, which causes $B+$ to fire. Due to the level expressions, no new rules are enabled when $B+$ fires. The rule r_3 is projected out of the matrix when it fires, leaving a matrix containing only a clock for r_5 . The firing of r_5 allows $C+$ to fire. Now, the rule r_7 , which enables $D+$, becomes enabled. It is now the only enabled rule, since r_5 is projected when it fires. In a similar manner, the firing of $D+$ generates a region containing a single rule which enables $x+$. All of the rules enabling falling transitions on $A+$, $B+$, $C+$, and $D+$ are waiting for x to become high. When x rises, r_2 , r_4 , r_6 , and r_8 become enabled at the same time, and their age differences in the matrix are set to zero. The region which is shown for this matrix is projected into the c_2, c_4 plane. It shows that c_2 and c_4 must be the same age and have a maximum age of 10. All of the downgoing transitions can now occur in any order. When A , B , C , and D , have fallen, x can fall, returning the TEL structure to its initial state.



3.5 Summary

The algorithm presented in this chapter allows us to find the state space of any TEL structure. It is a substantial improvement over the geometric algorithm presented by Rokicki in [58] since it can deal with multiple behavioral rules and boolean conditions. It can, however, generate a large number of regions since at least one region is generated for each firing sequence explored. The next chapters introduce the POSET algorithm, which dramatically reduces the number of regions needed to represent the timed state space.

CHAPTER 4

POSET TIMING I

The distinction between past, present, and future is only a stubbornly persistent illusion.
- Albert Einstein

While the geometric algorithm described in the previous chapter eliminates the single behavioral rule restriction and analyzed specifications with level expressions, Rokicki [59] and Bozga [13] show that the number of geometric regions the algorithm generates can explode for highly concurrent timed systems. In [59], an algorithm is described that uses *partially ordered sets* (POSETs) instead of linear sequences during state space exploration to mitigate this state explosion problem. POSET timing techniques take advantage of the inherent concurrency in the specification and prevents additional regions from being added for different sequences of firings that allow the same set of future behaviors in the system. This results in a compression of the state space into fewer, larger geometric regions that, taken together, contain the same region in space as the set of regions generated by the standard geometric technique.

The specifications used by Rokicki in [59] differ from TEL structures in two significant ways: they have the single behavioral rule restriction, and they do not include boolean expressions. In order to develop a POSET algorithm that can analyze TEL structures, both of these shortcomings must be dealt with. In this chapter, we present an algorithm that can analyze a class of TEL structure with multiple behavioral rules where all boolean expressions are **true**. In the next chapter, we extend the algorithm to analyze TEL structures with more interesting boolean expressions.

4.1 Creating Larger Equivalence Classes

The semantics described in Chapter 3 require two firing sequences to be in different equivalence classes if they result in the same set of enabled rules but allow different sets of values to be assigned to the active clocks. This is based on the observation that if two

sequences σ and σ' result in the same set of enabled rules, and allow the same set of values to be assigned to the active clocks, then a timed state is reachable from σ if and only if it is reachable from σ' . However, in some cases the requirement that the allowable clock values for both sequences must be the same is too restrictive. With additional analysis, it is possible to derive a set of clock values for a set of enabled rules, $enabled(\sigma)$, which are guaranteed to be allowed by some firing sequence σ' where $enabled(\sigma) = enabled(\sigma')$. In other words, given a firing sequence, σ , it is possible to determine not only which clock values are allowed for σ , but also a set of clock values that are guaranteed to be allowed for some other reachable firing sequence, σ' , in which concurrent events are fired in a different order. This allows the POSET algorithm to preemptively construct a larger region for σ , knowing that eventually a firing sequence, σ' , for which the clock values are allowed, is found during the depth first search. When σ' is found, the clock values that it allows are already represented in the region that is constructed for σ , and an additional region is not generated. This effectively combines the regions for σ and σ' and reduces the number of regions in the state space.

The computation necessary to determine this larger set of clock values is based on the causality in the sequence. The causal event function, E_c , which returns the event that is causal to a rule firing, is defined in Chapter 2. In order to develop the POSET algorithm, more definitions concerning causality are necessary. The first definition states that an event's causal rule firing is the rule firing immediately preceding it.

Definition 4.1.1 *The causal rule firing of an event firing σ_i is the rule firing immediately preceding it, σ_{i-1} .*

The rule firing immediately preceding σ_i is its causal rule. The firing of σ_{i-1} places $L(\sigma_i)$ in the firable set and therefore controls its firing time.

We can now define when an event firing σ_i is causal to another event firing σ_j .

Definition 4.1.2 *Event firing σ_i is causal to event firing σ_j ($causal(\sigma_i, \sigma_j, \sigma)$) if:*
 $\sigma_i = E_c(\sigma_{i-1}, \sigma)$.

An event firing σ_i is causal to event firing σ_j if σ_i is the causal event to σ_j 's causal rule. If σ_i is causal to σ_j then it is the firing time of σ_i that determines the firing time of σ_j .

Since this chapter assumes that all boolean expressions are **true**, if an event firing σ_i is causal to an event firing σ_j , there is always a rule connecting $L(\sigma_i)$ and $L(\sigma_j)$ (i.e

$\langle L(\sigma_i), L(\sigma_j), l, u, b, \rangle \in R \rangle$). The time separation between σ_j and its causal event firing σ_i is always less than the upper bound on this rule, u . This is formalized in the following lemma:

Lemma 4.1.1 *If σ_i is causal to σ_j in σ then the inequality: $\tau(\sigma_j) \leq \tau(\sigma_i) + u(\sigma_{j-1})$ is true for all valid timing assignments to σ .*

The proof of this lemma (as well as all following lemmas and theorems) is given in the appendix to this chapter. There is also a more general property that holds between any two event firings σ_i and σ_j . If the firing σ_i is the enabling event of a rule enabling σ_j , then the minimum time separation between the firings σ_i and σ_j is at least the lower bound on the rule.

Lemma 4.1.2 *If $L(\sigma_k) = \langle L(\sigma_i), L(\sigma_j), l, u, b \rangle \wedge i < k < j$ then the inequality $\tau(\sigma_j) \geq \tau(\sigma_i) + l(\sigma_k)$ is true for all valid timing assignments, τ , to σ .*

If all of the rules that enable the event fired by σ_i have empty choice sets, then the lower and upper bounds on these inequalities can always be met by some reordering of the firing sequence that is in Σ . In order to prove this, a few more definitions and lemmas are required. The first is the definition of the *required* set, which contains the set of firings in σ that must occur in order for a firing, σ_i , to meet the requirements specified by Definition 2.2.6(1) and (2). If σ_i is the first firing of an initially marked rule, then the required set of σ_i is empty. If σ_i is a rule firing, and is not the first firing of an initially enabled rule, then its required set contains its enabling event. If σ_i is an event firing, then firings of all of the rules that enable it are required for it to fire. If σ_i is an event firing, and σ_j is the enabled event of a rule whose enabling event is σ_i , then σ_j is in the required set of σ_i . This requirement follows from the one-safe property. When a rule is enabled by the firing of its enabling event, its enabled event must fire before its enabling event can fire again. The last condition is the transitive closure of these requirements, if a firing σ_j is required for σ_i then all events required for σ_j are also included in the required set for σ_i . These requirements are defined formally as follows:

Definition 4.1.3 *The required set of σ_i in $\sigma_{0..n}$ ($required(\sigma_i, \sigma_{0..n})$) is defined recursively as follows:*

1. $L(\sigma_i) = r \in R_0 \wedge \neg \exists \sigma_j \in \sigma_{0..i} : L(\sigma_j) = L(\sigma_i) \Rightarrow required(\sigma_i, \sigma_{0..n}) = \emptyset$
2. $L(\sigma_i) = r \in R \wedge (L(\sigma_i) \notin R_0 \vee \exists \sigma_j \in \sigma_{0..i-1} : L(\sigma_j) = L(\sigma_i)) \Rightarrow$
 $E_c(\sigma_i, \sigma_{0..n}) \in required(\sigma_i, \sigma_{0..n}).$
3. $L(\sigma_i) = e \in E \wedge L(\sigma_j) = \langle e', e, l, u, b \rangle \wedge (\neg \exists \sigma_k \in \sigma_{j+1..i} : L(\sigma_k) = \langle e', e, l, u, b \rangle \vee$
 $(L(\sigma_k) = \langle e', f, l, u, b \rangle \wedge f \# e)) \Rightarrow \sigma_j \in required(\sigma_i, \sigma_{0..n}).$
4. $L(\sigma_i) = e \in E \wedge L(\sigma_j) = f \wedge \langle e, f, l, u, b \rangle \in R \wedge i > j \Rightarrow \sigma_j \in required(\sigma_i, \sigma_{0..n})$
5. $\sigma_i \in required(\sigma_j, \sigma_{0..n}) \wedge \sigma_j \in required(\sigma_k, \sigma_{0..n}) \Rightarrow \sigma_i \in required(\sigma_k, \sigma_{0..n})$
(Transitive closure.)

A sequence σ' which is created from σ by changing the firing order is referred to as a *reordering* of σ . The reordering is described using a reordering function ρ . When ρ is given a firing sequence, σ , $\rho(\sigma)$ returns a new firing sequence which has the same firings occurring in a different order. When ρ is given a firing σ_i , $\rho(\sigma_i)$ returns the firing number of this firing in the reordered firing sequence. A sequence σ' is the result of a reordering $\rho(\sigma)$ if and only if $\forall \sigma_i \in \sigma : (\rho(\sigma_i) = x \Rightarrow \sigma'_x = \sigma_i)$. A firing $\sigma_i \in \sigma$ is equal to a firing $\sigma'_x \in \sigma'$ if $L(\sigma_i) = L(\sigma'_x)$, and they are both the same instance of $L(\sigma_i)$ in their respective sequences. It can be shown that if ρ meets the following conditions, then $\rho(\sigma) \in \Sigma$ if σ in Σ . The first requirement is that if σ_j is in the required set of σ_i , then σ_j cannot be made to fire after σ_i in the new sequence. The second requirement is that if a rule firing σ_{i-1} is followed by an event firing σ_i , then σ_{i-1} and σ_i are also consecutive in the reordering. The third requirement deals with choice. If σ_i is the firing of a rule with a non-empty choice set, then its enabled event may or may not fire following it. To determine this from the sequence, we define a function $next(\sigma_i, \sigma, e)$ which returns the next event enabled by e that fires after σ_i . Suppose that $L(\sigma_i) = \langle e, f, l, u, b \rangle$ and $next(\sigma_i, \sigma, e) = f$. In order for the reordered sequence to be valid, all of the firings that occur between σ_i and f , cannot be reordered arbitrarily. Once σ_i fires, no rule firing, σ_k , which enables an event that conflicts with f can be reordered to occur before a rule firing σ_j whose enabled event is f . This restriction is necessary to make sure that choices are not resolved differently in the reordered firing sequence and the original firing sequence. If the $next(\sigma_i, \sigma, e) \neq f$ then σ_i must not be reordered to occur after the event in its choice set that fires instead

of f . If σ_i is reordered after the firing of the event in its choice set, then $L(\sigma_i)$ would not be enabled when it fires since it loses its enabling when the event in its choice set fires. These conditions are defined formally as follows:

Definition 4.1.4 *A reordering ρ of σ is valid if:*

1. $\sigma_j \in \text{required}(\sigma_i, \sigma) \Rightarrow \rho(\sigma_j) < \rho(\sigma_i)$
2. $L(\sigma_i) = e \in E \Rightarrow \rho(\sigma_i) = \rho(\sigma_{i-1}) + 1$
3. $L(\sigma_i) = r = \langle e, f, l, u, b \rangle \in R \wedge \text{choice_set}(r) \neq \emptyset \wedge L(\sigma_m) = \text{next}(\sigma_i, \sigma, e) = f \Rightarrow$
 $\forall \sigma_j \in \sigma_{i+1..m}, \forall \sigma_k \in \sigma :$
 $(L(\sigma_j) = \langle e', f, l', u', b' \rangle) \wedge (L(\sigma_k) \in \text{choice_set}(r)) \Rightarrow \rho(\sigma_j) < \rho(\sigma_k)$
4. $L(\sigma_i) = r = \langle e, f, l, u, b \rangle \in R \wedge \text{choice_set}(r) \neq \emptyset \wedge L(\sigma_m) = \text{next}(\sigma_i, \sigma, e) \neq f \Rightarrow$
 $\rho(\sigma_i) < \rho(\sigma_m)$

The next lemma states that, if a sequence σ is in Σ , then any reordering of σ , $\rho(\sigma)$ is also in Σ .

Lemma 4.1.3 *Given $\sigma \in \Sigma$ and ρ is a valid reordering of σ , if $\sigma' = \rho(\sigma)$ then $\sigma' \in \Sigma$.*

Lemma 4.1.3 can be used to redefine what it means for two sequences to have the same timed state. Previously two sequences, σ and σ' , are defined to result in the same timed state if every set of clock ages that could result from a valid timing assignment to σ could also result from a valid timing assignment to σ' . The definition of a valid timing assignment is based on the concept of assigning firing times to rules and events that fired in sequence, and therefore must assign firing times that are consistent with the order that rules and events fire in the sequence. Timing assignments that allow rules and events to fire out of order can be made if it is guaranteed that a sequence exists that can fire in order with that timing assignment. The set of valid reorderings of a sequence σ defines when such a reordering exists by creating a partial order to which all of the sequences that can result from reordering σ must conform.

More formally, a sequence σ is used to define a partial order as follows.

Definition 4.1.5 *A partial order consists of a set (α) and an ordering relationship ($>$).*

The partial order defined by a sequence σ is as follows:

1. $\alpha = \{\sigma_i \in \sigma\}$

2. $> = \sigma_i > \sigma_j$ if and only if $\forall \rho(\sigma) : (\rho \text{ is valid} \Rightarrow \rho(\sigma_i) > \rho(\sigma_j))$

Two firings are only ordered in the partial order if they always occur in the same order for all valid reorderings of σ .

The set of firing sequences that can be derived by reordering the firings in σ in a way that conforms to the partial order defined by σ is referred to as $PO(\sigma)$. This set can be used to define a new set of valid timing assignments for σ .

Definition 4.1.6 *A timing assignment τ is PO valid for σ if $\exists \sigma' \in PO(\sigma) : \tau$ is valid for σ' .*

Two sequences σ and σ' can now be considered partial order equivalent if $enabled(\sigma) = enabled(\sigma')$ and τ is a PO valid timing assignment to σ if and only if there is a PO valid timing assignment τ' to σ' such that $\forall r_i \in enabled(\sigma) : \tau(c_i) = \tau'(c_i)$. This definition eliminates the ordering of concurrent events from consideration in creating the equivalence class, and therefore allows the equivalence classes to be larger. When a sequence σ is explored, a geometric region is created that includes all of the timing assignments that are PO valid for σ . A timing assignment is only PO valid for σ if there is some untimed reachable firing sequence for which it is valid. Therefore, even though a PO valid timing assignment may violate the ordering of σ , it is guaranteed that the search eventually finds a firing sequence for which it is valid. When this sequence is explored, the search can immediately backtrack, thus eliminating timed states.

In order to be able to build this larger region based on the partial order implied by a firing sequence, the algorithm must know what timing assignments are PO valid for σ while σ is being explored. Lemmas 4.1.1 and 4.1.2 show that there are upper and lower bounds on the separation between event firing times that depend only on causality. If causality is preserved in a reordering of a firing sequence, these upper and lower bounds are preserved as well. Therefore if for all sequences σ' in $PO(\sigma)$, $causal(\sigma, \sigma_i, \sigma_j) \Rightarrow causal(\sigma', \sigma'_{(\rho(\sigma_i))}, \sigma'_{(\rho(\sigma_j))})$, then all valid timing assignments to sequences in $PO(\sigma)$ satisfy the inequalities in the lemmas. The next lemma states that causality is preserved by reordering.

Lemma 4.1.4 *If $causal(\sigma, \sigma_i, \sigma_j)$ and ρ is a valid reordering used to map σ to σ' , then $causal(\sigma', \sigma'_{(\rho(\sigma_i))}, \sigma'_{(\rho(\sigma_j))})$*

If the geometric regions representing valid timing assignments are created based on Lemmas 4.1.1 and 4.1.2, then the entire state space is found, but it may contain invalid timing assignments since the lemmas do not guarantee that there are valid timing assignments that fall in the entire range allowed by the inequalities. This means that although all states in the state space are found, some extra states may be found as well. This may result in false negative verification results or suboptimal synthesized circuits. In order to explore the state space exactly, we need to be able to determine from the sequence σ , the minimum value of x and the maximum value of y , for which if σ_i is causal to σ_j , there exists a valid reordering of σ , such that $x < \tau(\rho(\sigma_i)) - \tau(\rho(\sigma_j)) \leq y$. Lemmas 4.1.1 and 4.1.2 provide bounds for these values and if σ_i is not enabled by any rules with non-empty choice sets, x and y are exactly the bounds from Lemmas 4.1.1 and 4.1.2.

Theorem 4.1.1 *For any firing sequence $\sigma \in \Sigma$ that has a valid timing assignment, if σ_i is causal to σ_j , and $L(\sigma_j)$ is not enabled by any rules with non-empty choice sets, there exists a firing sequence $\sigma' \in \Sigma$ created from a reordering ρ for which there is a valid timing assignment τ' where $\tau'(\sigma'_{(\rho(\sigma_i))}) + u(\sigma_{j-1}) = \tau'(\sigma'_{(\rho(\sigma_j))})$.*

This theorem means that if an event firing σ_i is causal to event firing σ_j and the event fired by σ_j is not enabled by any rules with non-empty choice sets, then the maximum separation between firings σ_i and σ_j over all valid reorderings of the sequence is defined. There is always a reordering with a valid timing assignment where the age of the clock associated with σ_j 's causal rule reaches its upper bound. Therefore there is always a reordering where the maximum separation between σ_i and σ_j is $u(\sigma_{j-1})$. This means it is possible to determine the maximum separation between σ_i and σ_j over all valid firing sequences where σ_i is causal to σ_j by examining a single firing sequence σ .

Theorem 4.1.2 *For any firing sequence $\sigma \in \Sigma$ that has a valid timing assignment, if σ_i is the firing of event e in σ , there exists at least one rule firing $\sigma_j : L(\sigma_j) = \langle e', e, l, u, b \rangle$ for which in some firing sequence $\sigma' \in \Sigma$ constructed from ρ there exists a valid timing assignment τ' in which $\tau'(\sigma'_{(\rho(E_c(\sigma_j, \sigma))})} + l(\sigma_j) = \tau'(\sigma'_{(\rho(\sigma_i))})$.*

This theorem deals with minimum separations between event firings. Unlike Theorem 4.1.1 it does not have the restriction that the event firing in question is not enabled by rules with non-empty choice sets. Intuitively, the theorem states that for every event firing σ_i , there exists a reordering with a valid timing assignment where σ_i fires at the

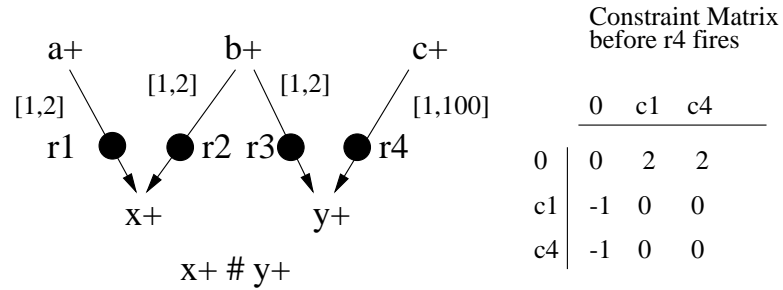


Figure 4.1. A choice computation.

minimum time allowed by the rules enabling it. This minimum time is the earliest time at which all of the rules enabling $L(\sigma_i)$ have clocks whose ages meet the lower bounds on the rules. The theorem shows that there is always a sequence where σ_i fires at this minimum time.

These theorems allow an algorithm to construct a region based state space representation for the set of timing assignments that are possible in a specification if it contains no conflicts. When there are conflicts, some rules have non-empty choice sets and the analysis becomes more complex. Although Theorem 4.1.2 still applies, Theorem 4.1.1 only applies to events that are not enabled by rules with non-empty choice sets. When a firing σ_j is enabled by a rule with a non-empty choice set, the maximum time separation between σ_j and its causal event σ_i may not be able to reach $u(\sigma_{i-1})$ for any valid reordering of σ . This is illustrated in Figure 4.1, where the rule $\langle b+, x+ \rangle$ has a choice set consisting of $y+$ and the rule $\langle b+, y+ \rangle$ has a choice set consisting of $x+$. Assume that $a+$, $b+$, and $c+$ all fire at time zero. If $c+$ is causal to $y+$, r_4 must fire after r_3 but before both r_1 and r_2 . If r_4 fires before r_3 , then r_3 is causal to $y+$. If r_4 fires after r_1 and r_2 fire, then $x+$ fires instead of $y+$. The event $c+$ can only be causal to $y+$ if r_4 fires between one and two time units after it becomes enabled, and it cannot reach its upper bound, 100. It is possible to compute the upper bound for events enabled by rules with non-empty choice sets, but the computation is complex, and in the worst case can involve examining the entire firing sequence. Therefore, when an event, e_i , which is enabled by a rule with a non-empty choice set fires, the maximum separation between e_i and its causal event is set to the maximum allowed by the *current* firing sequence. This means that all timing assignments to the firing of e_i that are in the region are valid for the current firing sequence. Therefore

no reordering of the rule firings which enable e_i is needed for e_i to fire at the computed upper bound. This ensures that the resulting region is exact, but the restriction results in more regions being generated than may be necessary.

The result of the restriction on reorderings imposed by rules with non-empty choice sets is that the worst-case complexity of the POSET algorithm, when applied to TEL structures with choice, is no better than the geometric algorithm presented in Chapter 3. However, in practice most circuit specifications are dominated by concurrent behavior rather than choice behavior. The POSET algorithm still shows significant benefit over the geometric algorithm in such a specification. In a specification consisting mostly of choice behavior, concurrency is limited and therefore state explosion is less of a problem. In this kind of specification the POSET algorithm essentially reduces to the geometric algorithm with some additional overhead. Alternatively, the geometric algorithm can be used directly on such a specification. Finally, we have found that for most circuit specifications, the additional restriction imposed by choice has little impact on the generated state space. If the restriction is eliminated, larger regions are generated, which are supersets of the actual regions, but new untimed states are rarely found. Therefore, eliminating the restriction may produce a conservative solution faster. If this is acceptable, events enabled by rules with non-empty choice sets can be treated the same as other events.

4.2 POSET Algorithm

The POSET algorithm creates the larger equivalence classes discussed in the previous section by maintaining a *POSET* matrix in addition to the constraint matrix discussed in Chapter 3. The POSET matrix stores the minimum and maximum possible separations between event firing times that can still effect future behavior. These separations represent the set of possible timing assignments to the partial order that is created by the firing sequence currently being explored. At each iteration, the separations in the POSET matrix are copied into the entries of the constraint matrix that restrict the differences in the ages of the rules. Events are projected out of the POSET matrix when their timing information is no longer needed, so the algorithm only needs to retain and operate on local timing information.

When a new event fires and is added to the POSET matrix, the minimum and maximum time separations between its firing time and the firing times of all other events in the matrix are determined. They must only allow timing assignments to the partial

order that are valid. This means that the separations must be consistent with the causality in the firing sequence being explored. This is the major difference between the POSET technique described here and the work presented by Rokicki in [58, 59]. In [58, 59], it is not necessary to use explicit causality information since the causal rule is always the behavioral rule. With multiple behavioral rules, causality must be considered in order to compute a correct POSET matrix.

Figure 4.2 shows the modified update algorithm from Chapter 3 which uses the POSET method. It calls an algorithm to update the POSET when an event fires. It then advances time by projecting out the firing rule, setting all of the maximum ages to the rule maximums, recanonicalizing, and normalizing

Figure 4.3 shows the algorithm which updates the POSET matrix, PM. The algorithm first examines all of the events currently in the POSET matrix and determines what relationship each event has to the firing event, f . This is simple since all of the information necessary to do this is easily stored as the firing sequence is being explored. If an event e_i in PM is the causal event for the firing event f , then the minimum separation between e_i and f in PM is set to the lower bound on r_c , f 's causal rule. If f is enabled by a rule with a non-empty choice set, then the maximum separation is set to the maximum age of r_c that is allowed by the current constraint matrix. This sets the separation to the maximum allowed by the current firing sequence, not over all valid reorderings of the current sequence. With this restriction, when an event which is enabled by a rule with a non-empty choice set fires, the maximum timing assignment that it can have is limited by the maximum amount time can advance before another rule must fire.

For example, consider the choice in Figure 4.1 and assume that events $a+$, $b+$, and $c+$ all fire at the same time. The constraint matrix that results after r_2 and r_3 fire is shown in the figure. If the rule r_4 fires next, the event $y+$ fires. Event $c+$ is causal to $y+$, and r_4 is the causal rule. The maximum bound on r_4 is 100, but this is not the value placed into the POSET matrix by the algorithm. Since $y+$ is enabled by a rule with a non-empty choice set, the value 2, which is the maximum age of r_4 in the *current* constraint matrix, is used instead.

Returning to the algorithm, if the firing event is only enabled by rules with empty choice sets, then the only limitation is the upper bound on the causal rule, and the separation between f and e_i is set to the upper bound on r_c . If an event is not causal, but does enable one of the rules that enables f , then a constraint is added indicating that

Algorithm 4.2.1 (Update)

```

void update(TEL structure TEL  $\langle N, s_0, A, E, R, R_0, \# \rangle$ , geometric region M,
           POSET matrix PM, rule  $r = \langle e, f, l, u, b \rangle$ , rule set  $R_{en}$ , bool event_fired) {
  if(event_fired) then
    update_POSET(TEL, PM, M, r,  $R_{en}$ );
  project(M, index(r));
  forall( $r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en}$ )
     $M[0][\text{index}(r_i)] = u_i$ ;
  recanonicalize(M);
  normalize(M);
}

```

Figure 4.2. Update the geometric region using POSETs.**Algorithm 4.2.2 (Update POSET)**

```

void update_POSET(TEL structure TEL  $= \langle N, s_0, A, E, R, R_0, \# \rangle$ , POSET matrix PM,
                 constraint matrix M, causal rule  $r_c = \langle e, f, l, u, b \rangle$ , rule set  $R_{en}$ ) {
  forall( $e_i \neq f : e_i$  is represented in PM)
    if( $e_i = e$ ) then {
       $PM[\text{index}(e_i)][\text{index}(f)] = -l$ ;
      if ( $\exists r_j = \langle e_j, f, l_j, u_j, b_j \rangle \in R : \text{choice\_set}(r_j) \neq \emptyset$ ) then
         $PM[\text{index}(f)][\text{index}(e_i)] = M[0][\text{index}(r_c)]$ ;
      else  $PM[\text{index}(f)][\text{index}(e_i)] = u$ ;
    } elseif( $\exists r_i = \langle e_i, f, l_i, u_i, b_i \rangle \in R$ ) {
       $PM[\text{index}(e_i)][\text{index}(f)] = -l_i$ ;
       $PM[\text{index}(t_f)][\text{index}(t_i)] = \infty$ ;
    } else {
       $PM[\text{index}(t_i)][\text{index}(t_f)] = \infty$ ;
       $PM[\text{index}(t_f)][\text{index}(t_i)] = \infty$ ;
    }
  }
  recanonicalize(PM);
  forall ( $e_i : e_i$  is represented in PM)
    if ( $\neg \exists r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en}$ ) project(PM, index}(e_i));
  project(M, index}(r_c));
  forall( $r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en}$ ) {
     $M[\text{index}(r_i)][0] = 0$ ;
    forall( $r_j \in R_{en}$ ) {
       $M[\text{index}(r_i)][\text{index}(r_j)] = PM[\text{index}(causal(r_i))][\text{index}(causal(r_j))]$ ;
    }
  }
}

```

Figure 4.3. Procedure for updating the geometric region.

the lower bound on the rule must be met, but the upper bound is left unconstrained by setting it to ∞ . If an event is unrelated to the firing event then both the minimum and maximum bounds are set to ∞ . Once all of the constraints have been added to the POSET matrix, it is recanonicalized, causing all of the unconstrained entries to be set to the maximum value allowed by the constraints. The process of updating the POSET matrix is completed by removing any events that no longer enable rules in R_{en} from the matrix.

The constraints computed in the POSET matrix can then be used to compute a new constraint matrix when an event fires. The minimum age of each rule is set to 0 since information about minimums is already included in the POSET matrix. Next, the algorithm sets each entry in the constraint matrix, which represent age differences between rules, to the time separation between their causal events. When the resulting constraint matrix is recanonicalized, some of the inequalities that are copied from the POSET matrix may be constrained further since the POSET inequalities do not take into account the fact that the age of a rule may not exceed its upper bound.

4.3 Example

Figure 4.4 shows timing analysis based on POSETs applied to the small TEL structure shown at the top of the figure. This example shows how the algorithm solves two of the problems that occur when using geometric regions for timed state space exploration: region splitting and multiple behavioral rules. In this example, initially the list of rules that can fire, RL , contains r_2 and r_1 . The POSET matrix contains a single event, $A+$. The constraint matrix shows that the maximum age of both r_2 and r_1 is five. From this timed state, either rule can fire. In this example, r_2 is chosen. The POSET matrix now contains the minimum and maximum separations between the firing times of $A+$ and $B+$. The values are copied into the constraint matrix, since they correspond to the age difference between rules enabled by $A+$ and rules enabled by $B+$. After the all pairs shortest path algorithm is run, the separation of 7 that is possible between the firing of $A+$ and the firing of $B+$ is reduced to 5 since the rule r_1 has a maximum bound of 5. In this state r_1 can fire, or r_4 can fire. Rule r_1 is chosen to fire next. When $C+$ fires, the POSET matrix no longer needs to contain $A+$ since it no longer enables any enabled rules. The POSET matrix shows that $B+$ could have fired at most 5 time units after $C+$ and $C+$ could have fired at most 2 time units after $B+$. Now there are three enabled

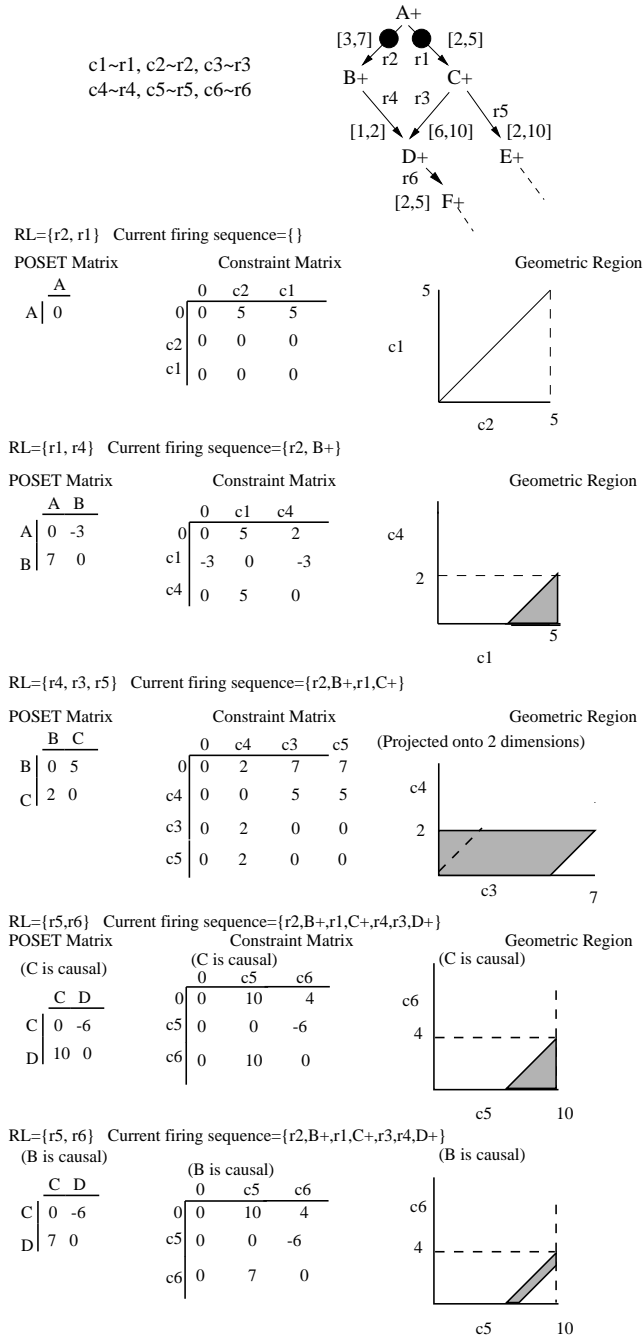


Figure 4.4. Example of timing with partially ordered sets.

rules and the region is 3-dimensional. In the figure, a two dimensional projection of the region onto the (c_3, c_4) plane is shown. This region shows the advantage of the POSET technique. Even though in this particular firing sequence $B+$ fires before $C+$, the region produced here contains timing assignments where $C+$ fires before $B+$. Since $B+$ and $C+$ occur in parallel, all of these timing assignments are valid for the partial order created by the firing sequence $[r_2, B+, r_1, C+]$. The dashed line in the middle of the region shows the two regions that would be generated by the standard geometric technique. The upper region contains timing assignments where $B+$ fires first, and the lower region contains timing assignments where $C+$ fires first. In this timed state, rules r_4, r_3 , and r_5 can fire. Once the rules r_4 and r_3 have fired, $D+$ fires. When $D+$ fires, information on event $B+$ can be removed from the POSET matrix, but since $C+$ still is the enabling event for an enabled rule, r_5 , $C+$ remains. Two different maximum separations between $C+$ and $D+$ are possible depending on whether event $C+$ or $B+$ is causal to $D+$. This is determined by whether the rule r_4 or r_3 fires last. The figure shows the two different geometric regions that result from the two different firing sequences. In this example, one region is a subset of the other, but this is not always the case.

4.4 Summary

Although the POSET algorithm presented in the chapter does not improve on the worst case complexity of the geometric algorithm from Chapter 3, it produces a huge performance improvement over the geometric algorithm when it is applied to highly concurrent examples. In some cases, as shown in Chapter 9, the improvement is many orders of magnitude. So far the POSET algorithm can only be applied to specifications without level expressions. The next chapter extends the benefits of the POSET algorithm to specifications with level expressions.

4.5 Appendix

Lemma 4.1.1 *If σ_i is causal to σ_j in $\sigma_{0..n}$ then the inequality: $\tau(\sigma_j) \leq \tau(\sigma_i) + u(\sigma_{j-1})$ is true for all valid timing assignments to $\sigma_{0..n}$.*

Proof: We know that the firing of event σ_i enabled the rule whose firing causes event σ_j to fire. This allows us to prove the desired inequality, $\tau(\sigma_j) \leq \tau(\sigma_i) + u(\sigma_{j-1})$.

$$\begin{aligned} \tau \text{ is valid} &\Rightarrow \tau(\sigma_{j-1}) \leq \tau(E_c(\sigma_{j-1}, \sigma)) + u(\sigma_{j-1}) \quad \{\text{Definition 2.2.8}\} \\ &\Rightarrow \tau(\sigma_{j-1}) \leq \tau(\sigma_i) + u(\sigma_{j-1}) \quad \{\sigma_i = E_c(\sigma_{j-1}, \sigma), \text{Definition 4.1.2}\} \\ &\Rightarrow \tau(\sigma_j) \leq \tau(\sigma_i) + u(\sigma_{j-1}) \quad \{\text{Definition 2.2.8, If } \tau \text{ is valid, } \tau(\sigma_j) = \tau(\sigma_{j-1})\} \end{aligned}$$

■

Lemma 4.1.2 *If $L(\sigma_k) = \langle L(\sigma_i), L(\sigma_j), l, u, b \rangle \wedge i < k < j$ then the inequality $\tau(\sigma_j) \geq \tau(\sigma_i) + l(\sigma_k)$ is true for all valid timing assignments, τ , to σ .*

Proof:

$$\tau \text{ is valid} \wedge (\sigma_i = E_c(\sigma_k, \sigma)) \Rightarrow \tau(\sigma_k) \geq \tau(\sigma_i) + l(\sigma_k) \quad \{\text{Definition 2.2.8}\} \quad (4.1)$$

Now we need to show that $\tau(\sigma_j) \geq \tau(\sigma_k)$ in order to prove the inequality. There are two cases to consider. The first is if σ_i is causal to σ_j in σ .

$$\begin{aligned} \text{causal}(\sigma, \sigma_i, \sigma_j) &\Rightarrow \sigma_k = \sigma_{j-1} \quad \{\text{Definition 4.1.2}\} \\ \sigma_k = \sigma_{j-1} &\Rightarrow \tau(\sigma_j) = \tau(\sigma_k) \quad \{\text{Definition 2.2.8}\} \\ \tau(\sigma_j) = \tau(\sigma_k) &\wedge (4.1) \Rightarrow \tau(\sigma_j) \geq \tau(\sigma_i) + l(\sigma_k) \end{aligned}$$

The second case is when σ_i is not causal to σ_j in σ .

$$\begin{aligned} \neg \text{causal}(\sigma, \sigma_i, \sigma_j) \wedge \sigma_i = E_c(\sigma_k, \sigma) \wedge L(\sigma_k) = \langle L(\sigma_i), L(\sigma_j), l, u, b \rangle &\Rightarrow \\ &k < j - 1 \quad \{\text{Definition 4.1.2}\} \\ k < j - 1 &\Rightarrow \tau(\sigma_k) \leq \tau(\sigma_j) \quad \{\text{Definition 2.2.8 and Definition 2.2.6}\} \\ \tau(\sigma_k) \geq \tau(\sigma_j) &\wedge (4.1) \Rightarrow \tau(\sigma_j) \geq \tau(\sigma_i) + l(\sigma_k) \end{aligned}$$

■

Before we can prove Lemmas 4.1.3 and 4.1.4 from the text, we need to prove two support lemmas concerning reorderings. The first lemma proves that a firing of a rule r

cannot be reordered to occur after a later firing of r in the sequence. The second lemma proves that an event cannot be reordered to occur after any future firings of rules that enable it.

Lemma 4.4.1 *Given that $\sigma \in \Sigma$, ρ is a valid reordering of σ , and $L(\sigma_i) \in R$:
 $(L(\sigma_i) = L(\sigma_j) \wedge i < j) \Rightarrow \rho(\sigma_i) < \rho(\sigma_j)$.*

Proof: Since $L(\sigma_i) = L(\sigma_j)$, they share the same enabling event and the same enabled event. Definition 4.1.3(4) places the firing of σ_i 's enabled event in the required set of σ_j 's enabling event. Since σ_i is in the required set of its enabled event, σ_j 's enabling event is in the required set of σ_j , and required sets are transitively closed, σ_i is in the required set of σ_j . Therefore σ_i cannot be reordered to occur after σ_j . ■

Lemma 4.4.2 *Given that $\sigma \in \Sigma$, ρ is a valid reordering of σ , and $L(\sigma_i) = e \in E$:
 $(L(\sigma_i) = L(\sigma_j) \wedge i < j) \Rightarrow \rho(\sigma_i) < \rho(\sigma_j)$.*

The rules that enable $L(\sigma_i)$ and $L(\sigma_j)$ are in their required sets. Since $L(\sigma_i) = L(\sigma_j)$ these rules are identical. Since their enabling rules are identical, their enabling rules are ordered by Lemma 4.4.1. Since the rules in the required set of σ_i are required by the rules in the required set of σ_j , σ_i is in the required set of σ_j and they must remain ordered. ■

Now we can prove the rest of the lemmas from the chapter.

Lemma 4.1.3 *Given that $\sigma \in \Sigma$ and ρ is a valid reordering of σ , if $\sigma' = \rho(\sigma)$ then $\sigma' \in \Sigma$.*

Proof: We need to show that $\forall \sigma'_x \in \sigma'$, σ'_x meets the requirements for a sequence to be in Σ (Definition 2.2.6). First we deal with event firings. The only requirement on event firings is stated in Definition 2.2.6(2). All event firings must satisfy the requirement.

Case 1: $L(\sigma'_x) = e \in E$.

We need to show that $L(\sigma'_x) \in \text{firable}(\sigma'_{0\dots x-1})$. The definition of firable (Definition 2.2.5) for events has a single requirement. An event $L(\sigma_i)$ is in the firable set for σ_{i-1} if:

$$\begin{aligned} \forall r = \langle e, f, l, u, b \rangle \in R : \exists \sigma_i \in \sigma : L(\sigma_i) = r \wedge \sigma_i \text{ is usable} \vee \\ \exists \sigma_j \in \sigma : L(\sigma_j) = \langle e', f, l, u, b \rangle \wedge e' \# e \wedge \sigma_j \text{ is usable} \end{aligned}$$

Since $\sigma \in \text{Sigma}$, this condition is true for σ_i . It is also true for σ'_x if no rule firing that is needed to fire σ_i in σ , can be moved after σ'_x in σ' , and no event firing which would

cause a needed rule firing, σ_j , to become unusable before σ_i fires is moved between the firing of σ_j and σ_i . If a rule firing is needed to fire σ_i then it is in the required set of σ_i , therefore it fires before σ'_x by the definition of a valid reordering (Def. 4.1.4). There are two proof obligations to ensure that no event firing which would cause a needed rule firing, σ_j , to become unusable is moved between the firing of σ_j and σ_i .

1. If σ_j is a rule firing used to fire σ_i , and σ_k is an event firing which occurs after σ_i and would make σ_j unusable, then σ_k cannot be reordered to occur before σ_i . This is proved as follows:

Suppose that $L(\sigma_i) = e$ and $L(\sigma_j) = r = \langle e', e, l, u, b \rangle$. An event firing could make $L(\sigma_j)$ unusable if it is a firing of e or it is a firing of an event in the choice set of r . Suppose that σ_k is such an event firing. If $L(\sigma_k) = e$, then e' is one of its enabling events, if $L(\sigma_k) = f$ and f is in the choice set of r , e' is also one of its enabling events. There must be some firing of e' between σ_i and σ_k to enable $L(\sigma_k)$. Therefore the firing of e' which occurs between σ_i and σ_k is in the required set of σ_k . The firing σ_i is in the required set of any future firings of e' by Definition 4.1.3(4). The firing of e' is in the required set of σ_k , and σ_i is in the required set of the firing of e' , therefore, since required set are transitively closed, a σ_k cannot be reordered to occur before σ_i .

2. If σ_j is a rule firing used to fire σ_i , and σ_k is an event firing which occurs before σ_j and would make σ_j unusable, then σ_k cannot be reordered to occur after σ_j . This is proved as follows:

Suppose the $L(\sigma_i) = e$ and $L(\sigma_j) = r = \langle e', e, l, u, b \rangle$. An event firing could make $L(\sigma_j)$ unusable if it is a firing of e or it is a firing of an event in the choice set of r . Suppose that σ_k is such an event firing. If σ_k occurs before σ_j , there must be a firing of e' that occurs between σ_k and σ_j . This firing is needed in order to fire σ_j and is in the required set of σ_j . The firing of σ_k is in the required set of the firing of e' by Definition 4.1.3(4). Therefore σ_k is in the required set of σ_j and cannot be reordered to occur after σ_j .

This shows that $L(\sigma'_x) \in \text{firable}(\sigma'_{0\dots x-1})$. The next two cases deal with the requirements placed on rule firings by Definition 2.2.6

Case 2: $L(\sigma'_x) = r = \langle e, f, l, u, b \rangle \in R$: We need to show that r is always enabled when it fires (Definition 2.2.6(1)). The requirements for a rule to be enabled are given in Definition 2.2.3. We need to show that in the reordering there is always a firing of e to enable the rule that has not be used by another firing of r or by the firing of an event in r 's choice set. (Since we are assuming all boolean expressions are **true**, the boolean expression part of the enabled definition is always satisfied.) Formally, we need to show that:

$$\begin{aligned} & \exists \sigma'_y \in \sigma'_{0..n} : L(\sigma'_y) = e \wedge \\ & \neg \exists \sigma'_w \in \sigma'_{x+1..n} : L(\sigma'_w) = (\sigma'_x) \wedge \\ & \neg \exists \sigma'_z \in \sigma'_{x+1..n} : L(\sigma'_z) \in \text{choice_set}(L(\sigma'_x)) \end{aligned}$$

Since $\sigma \in \Sigma$ this condition is satisfied by all rule firings $\sigma_i \in \sigma$. In order to show that is is satisfied in σ' we need to show that σ_i 's causal event firing, σ_j , is not moved after σ_i , no firing σ_k where $L(\sigma_k) = L(\sigma_i)$ can be moved between the firing of σ_j and σ_i , and no firing σ_k where $L(\sigma_k) \in \text{choice_set}(L(\sigma_i))$ can be moved between σ_j and σ_i .

The reordering restriction on the firing of σ_i 's causal event, σ_j , is guaranteed by Definition 4.1.3(2). The firing of $E_c(\sigma_i, \sigma)$ is in the required set of of σ_i and therefore σ_j cannot be reordered after σ_i by the definition of reordering.

The reordering restriction on other firings of $L(\sigma_i)$ is guaranteed by Lemma 4.4.1 for all firings of $L(\sigma_i)$ except the one immediately preceding σ_i . This firing, which we will call σ_k can be moved between σ_j and σ_i without violating the requirement that rule firings remain ordered. However, since the TEL structure is one-safe this cannot happen. Since σ_k is the firing of $L(\sigma_i)$ immediately preceding σ_i , σ_j is the only firing of event $L(\sigma_j)$ that occurs between σ_k and σ_i . Since the TEL structure is one-safe, this implies that the firing of of σ_k must occur before the firing of σ_j since the firing σ_k is necessary before the enabled event of $L(\sigma_i)$ can fire, allowing a firing of its enabling event, $L(\sigma_j)$ to occur. Therefore, no rule firing other than the firing of $L(\sigma_i)$ can be reordered to occur between σ_j and σ_i .

The final restriction is that any $\sigma_k : L(\sigma_k) \in \text{choice_set}(L(\sigma_i))$ cannot be reordered between σ_j and σ_i . Any firings of σ_k have $L(\sigma_j)$ as an enabling event by the definition of *choice_set*. Therefore a firing of σ_k always has a firing of $L(\sigma_j)$ in its required set. Since event firings must remain ordered by Lemma 4.4.2, firings of σ_k separated from σ_i by more than one firing of $L(\sigma_j)$ are excluded. Now we consider a firing of σ_k that occurs after σ_i , separated by one firing of $L(\sigma_j)$. The firing of σ_i is necessary in order for

σ_i 's enabled event to fire again. $L(\sigma_i)$'s enabling event, $L(\sigma_j)$, cannot fire again until its enabled event fires. Therefore, σ_i must fire before $L(\sigma_j)$ can fire again, and it is in the required set of the next firing of $L(\sigma_j)$. This means it is also in the required set of σ_k , and σ_k cannot be reordered to fire before σ_i . Finally we consider the firing of σ_k that occurs before σ_i , separated from σ_i , by one firing of $L(\sigma_j)$ which is the actual σ_j . In this case σ_k is in the required set of σ_j since σ_k must fire in order to fire the event that $L(\sigma_j)$ enables. This event must fire before σ_j can fire and enable the rule again. Therefore σ_k is in the required set of σ_j and cannot be reordered to occur after it.

We now need to prove that the remaining condition from Definition 2.2.6 is met.

Case 3: $L(\sigma'_x) \in R$, we need to show that:

$$L(\sigma'_x) \in R \wedge \text{firable}(\sigma'_{0..x}) \neq \emptyset \Rightarrow L(\sigma'_{x+1}) \in \text{firable}(\sigma'_{0..x}) \quad (4.2)$$

If the firable set of the subsequence ending in σ_i is non-empty in σ , it is followed by an event firing σ_{i+1} . In a valid reordered sequence, any firing which has a non-empty firable set in σ is followed by an event in σ' . Therefore, if no rule firing that has an empty firable set in σ has a non-empty one in σ' , the requirement is satisfied. Now we need to show that in a valid reordering it is not possible for a rule firing to have an empty firable set in σ and a non-empty one in σ' . Event firings and their causal rule firings are reordered consecutively. Therefore, if a rule firing, σ_i has an empty firable set in σ , and the result of its reordering, σ'_x , has a non-empty firable set in σ' , then any event in the firable set of the reordered σ_i is not the same event that actually used the rule firing in σ . This can only occur if the sequence is reordered in such a way that choices are resolved differently in σ and σ' . The definition of a valid reordering (Definition 4.1.4(3) & (4)) forces all choices to be resolved in the same direction in σ in σ' . Therefore $L(\sigma'_i) \in P \wedge \text{firable}(\sigma'_{0..i}) \neq \emptyset \Rightarrow L(\sigma'_{i+1}) \in \text{firable}(\sigma'_{0..i})$ holds.

We have now shown that $\sigma' \in \Sigma$ ■

Lemma 4.1.4 *If causal($\sigma, \sigma_i, \sigma_j$) and ρ is a valid reordering used to map σ to σ' , then causal($\sigma', \sigma'_{(\rho(\sigma_i))}, \sigma'_{(\rho(\sigma_j))}$)*

Proof: Definition 4.1.4(2) states that $L(\sigma_i) \in E \Rightarrow \rho(\sigma_i) = \rho(\sigma_{i-1}) + 1$. Therefore σ_i 's causal rule firing is the same in both sequences. Now we just need to show that $\sigma_j = E_c(\sigma_{i-1}, \sigma) \Rightarrow \sigma'_{\rho(\sigma_j)} = E_c(\sigma'_{\rho(\sigma_{i-1})}, \sigma')$. If $\sigma_j = E_c(\sigma_{i-1}, \sigma)$, then it is in the required set of σ_i and cannot be reordered to fire later than σ_i . This satisfies the first

constraint of Definition 2.2.7. Now we need to show that no other event firing that enables σ_{i-1} can be mapped between σ_j and σ_{i-1} . This event firing would have to be a firing of $L(\sigma_j)$ since there are no boolean expressions. Since firings of the same event remain ordered, the only event firing that could move between σ_j and σ_{i-1} is the firing of $L(\sigma_j)$ immediately following σ_{i-1} . But, σ_i is the firing of an event enabled by σ_j , so it is in the required set of the next firing of $L(\sigma_j)$. Thus, the next firing of $L(\sigma_j)$ cannot be reordered to occur before σ_i . Therefore, there is no valid reordering of the firing sequence where $\sigma'_{\rho(\sigma_j)} \neq E_c(\sigma'_{\rho(\sigma_{i-1})}, \sigma')$. Therefore, $causal(\sigma', \sigma'_{\rho(\sigma_i)}, \sigma'_{\rho(\sigma_j)})$ is true for all valid reorderings. ■

In order to prove the two theorems, additional definitions are necessary. These definitions specify the latest and earliest times that firings can occur with a valid timing assignment:

Definition 4.9.1 *Define the max_valid timing assignment to an event firing σ_i recursively as follows:*

1. $L(\sigma_i) \in R \Rightarrow max_valid(\sigma_i) = \min(max_valid(E_c(\sigma_i, \sigma)) + u(\sigma_i), max_valid(\sigma_{i+1}))$
2. $L(\sigma_i) \in E \Rightarrow max_valid(\sigma_i) = max_valid(\sigma_{i-1})$

Definition 4.9.2 *Define the min_valid timing assignment to a firing σ_i recursively as follows:*

1. $L(\sigma_i) \in R \Rightarrow min_valid(\sigma_i) = \max(min_valid(E_c(\sigma_i, \sigma)) + l(\sigma_i), min_valid(\sigma_{i-1}))$
2. $L(\sigma_i) \in T \Rightarrow min_valid(\sigma_i) = min_valid(\sigma_{i-1})$

These definitions follow directly from the definition of a valid timing assignment. Events always fire simultaneously with their causal rule, so their minimum and maximum firing times are determined by the minimum and maximum firing times of this rule. The minimum and maximum firing times of a rule are determined by when the rule becomes enabled, and by the other firings surrounding it in the sequence. Since the firing order of the sequence must be reflected by the timing assignment, the maximum valid timing assignment to a rule firing is limited by the maximum valid timing assignments of all firings following it. For the minimum valid firing time, the rule cannot fire before the minimum valid firing time of all rules preceding it. These definitions allow us to prove

upper and lower bounds on the times between event firings that are possible over all valid reorderings of a firing sequence.

Theorem 4.1.1 *For any firing sequence $\sigma \in \Sigma$ that has a valid timing assignment, if σ_i is causal to σ_j , and σ_j does not conflict with any other event, there exists a firing sequence $\sigma' \in \Sigma$ for which there is a valid timing assignment τ' where $\tau'(\sigma'_{(\rho(\sigma_i))}) + u(\sigma_{j-1}) = \tau'(\sigma'_{(\rho(\sigma_j))})$.*

Proof: Definition 4.9.1 states that this equation can always be satisfied for any σ where σ_i is causal to σ_j unless there is some σ_k that limits the maximum firing time of σ_j . A firing σ_k limits that maximum firing time of σ_j if it fires after σ_j in σ and has a lower maximum valid firing time than σ_j . Since σ_j is an event, it must fire at the same time as its causal rule firing σ_{j-1} . All firings limiting the firing time of σ_j are actually limiting the firing time of σ_{j-1} and must be moved to fire before σ_{j-1} . We need to show that we can create a valid reordering ρ which generates a sequence where all such firings are moved before the firing of σ_{j-1} . Since σ_j is not enabled by a rule with a non-empty choice set, only requirement (1) of Definition 4.1.4 applies to the order of firings relative to σ_{j-1} . Therefore, we can move all $\sigma_k : \sigma_j \notin \text{required}(\sigma_k)$ before the firing of σ_{j-1} . We create a reordering ρ where:

$$\rho(\sigma_k) > \rho(\sigma_{j-1}) \Rightarrow (k = j) \vee \sigma_j \in \text{required}(\sigma_k) \vee \quad (4.3)$$

$$\text{max_valid}(\sigma_k) \geq \text{max_valid}(E_c(\sigma_{j-1}, \sigma)) + u(\sigma_{j-1}) \quad (4.4)$$

This implies the following in a sequence $\sigma' = \rho(\sigma)$ where $\rho(\sigma_j) = x$ and $\rho(\sigma_k) = y$:

$$y > x \Rightarrow \sigma'_x \in \text{required}(\sigma'_y) \vee \text{max_valid}(\sigma'_y) \geq \text{max_valid}(E_c(\sigma'_{x-1}, \sigma')) + u(\sigma'_{x-1}) \quad (4.5)$$

Any firing that occurs after σ'_x (which is the reordered σ_j) in the new sequence either did not limit the firing time of σ_j in σ or requires σ_j to fire. All firings that have σ'_x in their required sets can now be given timing assignments that do not limit the firing time of σ'_x because σ'_x must fire before they can fire, and moving its maximum valid timing assignment later also moves theirs later. Since no firings that limit the firing time of σ'_x occur after σ'_x , this can always be done without violating the ordering constraint. Therefore there exists a firing sequence $\sigma' \in \Sigma$ for which there is a valid timing assignment τ' where $\tau'(\sigma'_{(\rho(\sigma_k))}) + u(\sigma_{j-1}) = \tau'(\sigma'_{(\rho(\sigma_j))})$ ■

Theorem 4.1.2 *For any firing sequence $\sigma \in \Sigma$ that has a valid timing assignment, if σ_i is the firing of event e in σ , there exists at least one rule firing $\sigma_j : L(\sigma_j) = \langle e', e, l, u, b \rangle$ for which in some firing sequence $\sigma' \in \Sigma$ constructed from ρ there exists a valid timing assignment τ' in which $\tau'(\sigma'_{(\rho(E_c(\sigma_j, \sigma))})} + l(\sigma_j) = \tau'(\sigma'_{(\rho(\sigma_i))})$.*

Proof: The proof of this theorem is similar to the proof of the Theorem 4.1.1. The goal is to move any firings that are limiting the minimum firing time of σ_i to fire after σ_i . Since the firing time of an event is determined by the rule firing, we are again dealing with the firing time of the rule firing σ_{i-1} . Since this time we are trying to move firings to occur after σ_i instead of before σ_i , Definition 4.1.4(3) does not restrict the possible reorderings relative to σ_i . Also any firing that occurs after σ_i cannot be in the required set of any firing that occurs before σ_i since $\sigma \in \Sigma$. Therefore all firings that occur before σ_i that are not in the required set of σ_i and limit the minimum firing time of σ_i can be reordered to fire after σ_i . When this is done, only firings that are in the required set of σ_i limit its minimum firing time. Since all of the rule firings necessary to fire σ_i are in its required set, there is at least one rule for which σ_i can fire at its minimum firing time. ■

CHAPTER 5

POSET TIMING II

*Good order is the foundation of all things.
- Edmund Burke*

The previous chapter presents a version of the POSET algorithm that analyzes TEL structures with multiple behavioral rules. This version of the algorithm assumes that all boolean expressions in the TEL structure are **true**, which is clearly a severe limitation. Therefore, this chapter extends the algorithm further, to allow it to analyze TEL structures with non-trivial level expressions.

The algorithm in Chapter 4 is based on the ability to take a sequence $\sigma \in \Sigma$ and change the firing order so that a given timing assignment can be made to it. The chapter presents a number of conditions which must be met by the reordering in order to ensure that the new sequence is a valid firing sequence for the TEL structure. These reordering conditions do not consider the impact of boolean expressions. In order to modify the algorithm from the last chapter to work on TEL structures with boolean expressions, additional reordering conditions which consider the impact of boolean expressions are needed to guarantee that the reordered sequence preserves causality and conforms to the requirements in Definition 2.2.6.

5.1 Extending the Required Set

The first reordering restriction in Definition 4.1.4 orders firings based on required sets. If a firing σ_j is in the required set of σ_i , σ_j must fire before σ_i in the reordered sequence. This requirement still applies to TEL structures with boolean expressions, but the definition of required, Definition 4.1.3, needs to be extended to consider boolean expressions. The new definition of the required set is shown in Definition 5.1.1 below.

Definition 5.1.1 *The required set of σ_i in $\sigma_{0..n}$ ($required(\sigma_i, \sigma_{0..n})$) is defined recursively as follows:*

1. $L(\sigma_i) = r \in R_0 \wedge \neg \exists \sigma_j \in \sigma_{0..i} : L(\sigma_j) = L(\sigma_i) \Rightarrow required(\sigma_i, \sigma_{0..n}) = \emptyset$
2. $L(\sigma_i) = r \in R \wedge \neg (L(\sigma_i) \in R_0 \wedge \neg \exists \sigma_j \in \sigma_{0..i-1} : L(\sigma_j) = L(\sigma_i)) \Rightarrow E_c(\sigma_i, \sigma_{0..n}) \in required(\sigma_i, \sigma_{0..n})$.
3. $L(\sigma_i) = e \wedge L(\sigma_j) = \langle e', e, l, u, b \rangle \wedge (\neg \exists \sigma_k \in \sigma_{j+1..i} : L(\sigma_k) = \langle e', e, l, u, b \rangle \vee (L(\sigma_k) = \langle e', f, l, u, b \rangle \wedge f \# e)) \Rightarrow \sigma_j \in required(\sigma_i, \sigma_{0..n})$.
4. $L(\sigma_i) = e \wedge L(\sigma_j) = f \wedge \langle e, f, l, u, b \rangle \in R \wedge i > j \Rightarrow \sigma_j \in required(\sigma_i, \sigma_{0..n})$
5. $\sigma_i \in required(\sigma_j, \sigma_{0..n}) \wedge \sigma_j \in required(\sigma_k, \sigma_{0..n}) \Rightarrow \sigma_i \in required(\sigma_k, \sigma_{0..n})$
(Transitive closure.)
6. $L(\sigma_i) = \langle e, f, l, u, b \rangle \wedge L(\sigma_j) = e \wedge j < i \wedge \neg \exists \sigma_k \in \sigma_{j..i} : L(\sigma_k) = e \Rightarrow \sigma_j \in required(\sigma_i, \sigma)$
7. $L(\sigma_i) = r = \langle e, f, l, u, b \rangle \wedge r \text{ is disabling} \wedge \neg b(\phi(\sigma_{0..j-i, j+1..i-1})) \Rightarrow \sigma_j \in required(\sigma_i, \sigma)$
8. $L(\sigma_i) = r = \langle e, f, l, u, b \rangle \wedge r \text{ is non-disabling} \wedge \sigma_j = E_c(\sigma_i, \sigma_{0..n}) \wedge \neg b(\phi(\sigma_{0..k-1, k+1..j})) \Rightarrow \sigma_k \in required(\sigma_i, \sigma)$

The first five items in the definition are from the previous chapter and the last three are extensions. The first extension concerns enabling events. Definition 5.1.1(2) states that the causal event of a rule firing is in the required set of that rule firing. When there are no boolean expressions, this causal event is always the enabling event of the rule. When boolean expressions are added, the enabling event could be an event that causes the boolean expression to become true. If this is the case, the definition of required does not include the enabling event of every rule in its required set and this needs to be added. This is done formally in Definition 5.1.1(6).

Additions to the required set are also necessary to ensure that the boolean expression is satisfied when a rule fires. The causal event is in the required set, and this event may be the one whose firing causes the value of the boolean expression to change from false to true. However, all of the other event firings which are necessary to cause the boolean expression to evaluate to true are not included by the original definition. Consider for example the rule $e+ \rightarrow f+$ which has a boolean expression $a \wedge b$, and assume that it is enabled by the firing sequence $a+, e+, b+$. The original required definition includes only

the firing of $b+$ in the required set of $[e+, f+]$. Definition 5.1.1(6) includes the firings of $e+$ and $b+$ in the required set. The firing of $a+$ is not included by either the original definition or Definition 5.1.1(6), but it is needed in order for $[e+, f+]$ to be enabled when it fires. We need to add an additional condition to the required definition to deal with this. The condition is slightly different depending on whether the rule is disabling or non-disabling and defined formally in Definition 5.1.1(7) and (8). If the rule is disabling, the boolean condition must be true when the rule fires and all firings necessary for it to be true must be in the required set. If the rule is non-disabling, then the boolean expression only needs to be true when the rule becomes enabled. After the rule becomes enabled, the boolean expression can become false before the rule fires and the rule is still enabled when it fires. Definition 5.1.1(7) is for disabling rules. If a rule firing σ_i is a disabling rule, and the removal of another firing, σ_j , from the sequence would cause b to evaluate to false on the state generated by the sequence, then σ_j is in the required set of σ_i . Definition 5.1.1(8) is for non-disabling rules. The condition is similar, but it requires that the boolean expression is true when the rule is *enabled*, not when it fires. Firings that are included in the required set of σ_i due to the conditions in Definition 5.1.1(7) and (8) are referred to as the *context* set of σ_i , ($context(\sigma_i, \sigma)$), since their firings are required to create a boolean state in which σ_i can be enabled. These conditions include all firings needed for a rule to be enabled when it fires in the sequence in the required set.

5.2 Adding Reordering Restrictions

Unfortunately, the extension of the required set is not sufficient for Definition 4.1.4 to guarantee that any valid reordering of a sequence $\sigma \in \Sigma$ is also in Σ . Additions to the definition of a valid reordering are also necessary. The new definition of a valid reordering is shown in Definition 5.2.1 below:

Definition 5.2.1 *A reordering ρ of σ is valid if:*

1. $\sigma_j \in required(\sigma_i, \sigma) \Rightarrow \rho(\sigma_j) < \rho(\sigma_i)$
2. $L(\sigma_i) = e \in E \Rightarrow \rho(\sigma_i) = \rho(\sigma_{i-1}) + 1$
3. $L(\sigma_i) = r = \langle e, f, l, u, b \rangle \in R \wedge choice_set(r) \neq \emptyset \wedge L(\sigma_m) = next(\sigma_i, \sigma, e) = f \Rightarrow$
 $\forall \sigma_j \in \sigma_{i+1..m}, \forall \sigma_k \in \sigma :$
 $(L(\sigma_j) = \langle e', f, l', u', b' \rangle) \wedge (L(\sigma_k) \in choice_set(r)) \Rightarrow \rho(\sigma_j) < \rho(\sigma_k)$

4. $L(\sigma_i) = r = \langle e, f, l, u, b \rangle \in R \wedge \text{choice_set}(r) \neq \emptyset \wedge L(\sigma_m) = \text{next}(\sigma_i, \sigma, e) \neq f \Rightarrow \rho(\sigma_i) < \rho(\sigma_m)$
5. *If $\sigma_j \in \text{context}(\sigma_i, \sigma) \wedge ((L(\sigma_j) = x+ \wedge L(\sigma_k) = x-) \vee (L(\sigma_j) = x- \wedge L(\sigma_k) = x+))$ then $(k > i \Rightarrow \rho(\sigma_k) > \rho(\sigma_i)) \wedge (k < j \Rightarrow \rho(\sigma_k) < \rho(\sigma_j))$.*
6. *If $\sigma_j = E_c(\sigma_i, \sigma)$ and σ_k occurs before σ_j and the firing of σ_k would be causal to σ_i if it occurred after σ_j then $\rho(\sigma_k) < \rho(\sigma_j)$.*
7. *If $\sigma_j = E_c(\sigma_i, \sigma)$ and σ_k occurs after σ_j and the firing of σ_k would be causal to σ_i if it occurred before σ_j then $\rho(\sigma_k) > \rho(\sigma_j)$.*

The first four parts of the definition are from the previous definition of valid reordering. These requirements, along with the new required set definition do ensure that no firings needed to enable a rule are reordered to occur after it. However, they do not prevent events whose firing may disable a rule firing from being reordered to occur before its firing. Consider again the rule $e+ \rightarrow f+$ with boolean expression $a \wedge b$ and the firing sequence $e+, a+, b+, [e+, f+]$. All of the firings in the sequence are now in the required set of $[e+, f+]$, but there is nothing in the definition of reordering to prevent a firing of $a-$ from being reordered to occur before $[e+, f+]$ as follows: $e+, a+, b+, a-, [e+, f+]$. Now, assuming that $[e+, f+]$ is disabling, it is not enabled when it fires. This problem can also occur with non-disabling rules. Consider that the firing of $a-$ is reordered to occur before the firing of $b+$, creating the following firing sequence: $e+, a+, a-, b+, [e+, f+]$. In this sequence, the rule $[e+, f+]$ is never enabled at all since its boolean expression is never satisfied.

The first extension to the definition of valid reordering (item 5) ensures that this cannot happen. This addition to the reordering definition ensures that if a firing σ_j is context to σ_i , no firing which reverses the effect of σ_j is reordered to occur between σ_j and σ_i . This addition, along with the new required set, is sufficient to ensure that all reorderings of a sequence σ_i are valid. The addition is a bit overly restrictive since it prevents a reversing event from being reordered between a context event firing and the rule firing in all cases. For non-disabling rules there are some situations where the reversing event does not disable the rule. However, for algorithmic purposes it is simpler to assume that reversing events can never be reordered.

Another concern is the preservation of causality. When there are no boolean expressions, any reordering of the sequence where the rule firing immediately preceding each

event firing does not change, preserves the causality in the sequence. The last rule firing before an event, σ_i , fires is always the causal rule of σ_i by definition. If all boolean expressions are **true**, then the enabling event for the rule σ_{i-1} is always the causal event for event firing σ_i . With boolean expressions, this is not the case. Another event firing may have caused the rule that fires in σ_{i-1} to become enabled, by causing the current state to satisfy its boolean expression. A firing sequence σ must not be reordered in a way that changes the identity of this event. Therefore, the algorithm needs an additional reordering restriction that ensures that the identity of the causal event does not change. These restrictions are specified in items 6 and 7 of Definition 5.2.1.

We have now defined a set of restrictions on reordering to ensure that the reordered sequence is valid and that it preserves the causality of the original sequence. However, determining which reorderings meet these restrictions algorithmically is difficult when boolean expressions are complex. When only the restrictions from the previous chapter are used, the ability to reorder an event firing is independent of other reordering decisions and each event movement can be made independently. When arbitrary boolean expressions are included, the ability to change the firing order of σ_i and σ_j can depend on whether the location of another event σ_k has been changed. For example, consider the boolean expression $a \wedge (b \vee c)$ on a rule where the enabled event is $f+$. Suppose that in the original firing sequence a , b , and c are all true when $f+$ fires. Either $b+$ or $c+$ could be reordered to occur after $f+$ because doing so would not cause Definition 5.1.1 to be violated. However, once the decision has been made to reorder $b+$ after $f+$, $c+$ cannot be reordered after $f+$. Since reordering decisions are no longer independent, it is difficult to examine all possible reorderings at once since a reordering of one event may exclude the reordering of another event.

Therefore, if POSET timing is used, the TEL structures are limited to those where each boolean expression is either a single **and** term or a single **or** term. When this restriction is used reorderings can again be considered independently. In a simple **and** expression all context signals must remain before the rule firing. In an **or**, the only context signal firings are the causal event firing and the enabling event. Any other firings involved in the **or** expression can be reordered to occur after the rule firing. In practice, limiting the specification to single **ands** and **ors** does not prove to be a significant limitation. If a more complex boolean expression is required, the results from the POSET algorithm are conservative. If an exact result with arbitrary expressions is needed, then either

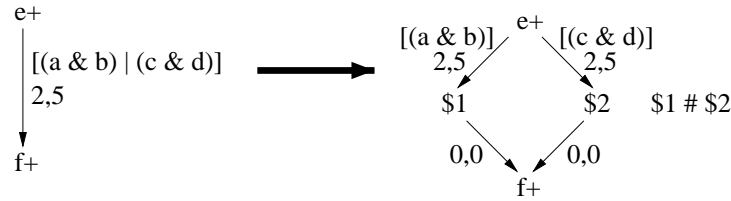


Figure 5.1. Transformations to simple **and** and **or**.

the simpler, geometric algorithm for TEL structures from Chapter 3 can be used, or the specification can be transformed into one that contains only simple **and** and **or** expressions.

Figure 5.1 shows the transformation. Before a graph can be transformed, all boolean expressions must be in sum of products form. Sequencing events are created for each of the terms and rules are added between the rule's enabling event and each of the sequencing events. Rules are also added between each of the sequencing events and the enabled event. Each of the rules leading from the enabling event is assigned one of the terms from the sum of products and the delay from the original rule. The rules enabled by the sequencing events are assigned a boolean expression of **true** and a delay of 0. All possible pairs of sequencing events are added to the conflict set. Only one of the sequencing events must fire in order to fire the enabled event. This transformation works without and algorithmic changes if the rules with complex boolean expressions are non-disabling. If the rules are disabling, the algorithm must keep know which rules are created by transformation and only generate a disabling error if all of them become disabled. This is necessary since the disabling of only one of the rules generated by the transformation does not indicate that the original rule is disabled. Although this transformation adds a significant number of extra rules and events for large expressions, large expressions are rare.

5.3 Simplified Restrictions

When the boolean expressions are restricted, simpler, more algorithmic versions of the additional reordering restrictions can be developed. For any firing $\sigma_i \in \sigma$:

1. If $L(\sigma_i)$ is a rule where $b = \mathbf{true}$, there are no additional restrictions.
2. If $L(\sigma_i)$ is an event, then there are no additional restrictions.

3. If $L(\sigma_i)$ is a disabling rule, no event which would disable $L(\sigma_i)$ can be moved before σ_i .
4. If $L(\sigma_i)$ is a non-disabling rule, and $\sigma_j = E_c(\sigma_i, \sigma)$, no event which would prevent the firing of σ_j from enabling σ_i can be moved before σ_j .
5. If $L(\sigma_i)$ is a rule with an **and** expression and σ_j is the causal event of σ_i then:
 - (a) The enabling event of $L(\sigma_i)$ cannot be moved after σ_j .
 - (b) No context firing can be moved after σ_j .
6. If $L(\sigma_i)$ is a rule with an **or** expression, and σ_j is the causal event to σ_i then:
 - (a) The enabling event of $L(\sigma_i)$ cannot be moved after σ_j .
 - (b) No firing which causes the **or** expression to become true can be moved before σ_j .
 - (c) If $L(\sigma_j)$ is the enabling event of $L(\sigma_i)$, then no event can be reordered to occur after σ_j if it would cause the **or** expression to be false when σ_j fires.

The first four conditions apply equally to **and** and **or** expressions. Obviously, if there is no boolean expression, then the old reordering restrictions that do not consider them are sufficient. If the firing is an event, there are no additional restrictions since all of the added conditions in the previous section concern rules. If a rule is disabling, a reordering should not cause a rule to become disabled if it does not do so in the original sequence. For example, if a disabling rule $e+ \rightarrow f+$ has a boolean expression $a \wedge b$, a firing of $a-$ cannot be reordered to occur before the rule firing. If the rule is non-disabling, the restriction is needed that no event prevents the rule's causal event from enabling it. For example, consider the non-disabling rule $e+ \rightarrow f+$, which has boolean expression $a \wedge b$, and causal event $e+$. If $e+$ is causal, then $a \wedge b$ is true when it fires. No firing of $a-$ can be moved to fire before $e+$ since it would not allow the firing of $e+$ to enable the rule.

There are specific additional restrictions for rules with **and**'s and **or**'s. If there is an **and** expression, then a reordering may change the causality or cause the new sequence to be invalid if some signal firing is moved later in the sequence. The restriction prevents a sequence from being created where the **and** expression is not true when the rule fires. Since no event firing which effects the expression may be moved after the causal event, it also ensures that the causal event for the rule firing remains the same. For example,

consider a rule $e+ \rightarrow f+$ which has a boolean expression $a \wedge b$, and assume that $b+$ is causal in the firing sequence. The fact that $b+$ is causal implies that there has been a firing of $e+$ and a firing of $a+$ somewhere in the sequence before $b+$. In a reordering, the firings of $e+$ and $a+$ are not allowed to be moved after $b+$ in the firing sequence.

With **or** expressions, three conditions are necessary. As with **and** expressions, the enabling event must not be reordered to occur after the causal event. The reordering also must ensure that the boolean expression does not become true too *early*. If the reordering moves an event firing that satisfies the **or** to occur before the causal event then the causality changes. Therefore, this is not allowed. For example, consider a rule $e+ \rightarrow f+$ which has a boolean expression $a \vee b$, and assume that $b+$ is causal in the firing sequence. The firing of $e+$ cannot move after $b+$ just like in the **and** expression. However, no firing of $a+$ can be allowed to move before $b+$ unlike in the **and** expression. If $a+$ fires first, then it is the causal event. The final condition ensures that the **or** expression is satisfied when the rule fires. If the enabling event is the causal event, the firing which satisfied the **or** expression cannot be moved after the firing of the enabling event. Arbitrary boolean expressions require combinations of these requirements which could be defined, but would be difficult to implement in an algorithm that is building geometric regions. The next section describes how these reordering conditions are used to build geometric regions which represent timing assignments to reorderings of the firing sequence.

5.4 Extended POSET Algorithm

This section describes how the new reordering restrictions are implemented in the POSET algorithm. The changes occur only in the function that updates the POSET matrix. Time separations that are left unbounded by the algorithm in the previous section are now assigned values to satisfy the new reordering restrictions.

Figure 5.2 shows the procedure for updating the POSET matrix. It has one argument that the update in the previous chapter does not, R_{used} . This set contains the rules that enable f_c , the firing event, and are in the fired set when f_c fires. These are the rule firings in the sequence that are used to fire f_c . Like in the previous chapter, each entry in the matrix represents the maximum time separation possible between two event firings over all possible valid reorderings of the firing sequence. When a new event, f_c , fires, entries must be added to store the separations between f_c and all of the other events represented

Algorithm 5.4.1 (Update the POSET)
 update_poset(*causal rule* $r_c = \langle e_c, f_c, l_c, u_c, b_c \rangle$, *used rule set* R_{used} , *POSET matrix* PM ,
constraint matrix M , *TEL* $\langle N, s_0, A, E, R, \# \rangle$, *state* s_c) {
 forall($e_i : e_i$ is represented in PM) {
 $PM[index(f)][index(e_i)] = \infty$;
 $PM[index(e_i)][index(f)] = \infty$;
 }
 forall($e_i : e_i$ is represented PM) {
 if($e_i = causal(r_c)$) **then** {
 if($\forall r = \langle e, f_c, l, u, b \rangle \in R : choice_set(r) = \emptyset$) **then**
 $PM[index(f_c)][index(e_i)] = u_c$;
 else $PM[index(f_c)][index(e_i)] = M[0][index(r_c)]$;
 if ($disable(e_i, f_c)$) **then**
 $PM[index(e_i)][index(f_c)] = 0$;
 forall($r = \langle e, f_c, l, u, b \rangle \in R_{used}$) {
 $e_{rc} = causal(r)$;
 if($e_i = e \wedge PM[index(f_c)][index(e_i)] > -l$) **then**
 $PM[index(e_i)][index(f_c)] = -l$;
 if($e_i = e_{rc} \wedge PM[index(e_f)][index(e_i)] > -l$) **then**
 $PM[index(e_{rc})][index(e_i)] = -l$;
 if($and_context(e_i, r) \wedge PM[index(e_{rc})][index(e_i)] > 0$) **then**
 $PM[index(e_{rc})][index(e_i)] = 0$;
 if($or_context(e_i, r) \wedge PM[index(e_i)][index(e_{rc})] > 0$) **then**
 $PM[index(e_i)][index(e_{rc})] = 0$;
 }
 }
 }
 recanonicalize(PM);
 forall ($e_i : e_i$ is represented in PM) {
 if ($\neg \exists r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en} \wedge \neg match(e_i, s_c)$) **then**
 project($PM, index(e_i)$);
 }
 forall($r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en}$) {
 $M[index(r_i)][0] = 0$;
 forall($r_j \in R_{en}$) {
 $M[index(r_i)][index(r_j)] = PM[index(causeal(r_i))[index(causeal(r_j))]$;
 }
 }
 }

Figure 5.2. Procedure for updating the POSET matrix.

in the matrix. The function first initializes all of the new entries in the matrix to infinity. A value of infinity means that there is no reordering restriction that applies to this event pair.

The rest of the algorithm checks the various reordering restrictions and changes the

values in the matrix accordingly. For each event e_i in the POSET matrix, the algorithm first determines if e_i is the causal event to the causal rule r_c . If e_i is the causal event and the firing event is not enabled by any rules with a non-empty choice set, then its firing time determines the upper bound on the firing time of f_c over all valid reorderings. This separation is thus set to the upper bound of the causal rule, u_c . If the firing event f_c is enabled by a rule with a non-empty choice set then, the upper bound in the POSET matrix is set to the upper bound on the causal rule in the constraint matrix. This sets the upper bound on the firing time of f_c to be the latest allowable by the *current* firing sequence. Then, the function checks if this event firing could disable a rule that enables the event in the POSET matrix that is currently being examined, e_i . If it does, then f_c must always occur after e_i , and their minimum separation is set to 0, indicating that f_c cannot occur before e_i .

The next step is to check all of the other reordering restrictions. Since the reordering restrictions are defined with respect to rule firings, the algorithm needs to apply the reordering restrictions to all of the rule firings that are used to fire the event f_c . First, the algorithm extracts the causal event for the rule that it is considering, e_{r_c} . In practice, it is simple to store the causal event of a rule when it becomes enabled. It then checks to see if e_i is an enabling event of r . If e_i is the enabling event of r , then the lower bound on r must be met for any valid reordering and the lower bound in the matrix is set to $-l$ if it is not already less than $-l$. The event e_i may also be the causal event of r , and this also implies the the minimum separation between e_i and f_c is l . Next, the algorithm checks for events that are required for an expression associated with r to be satisfied. Any such events must fire before the causal event, and therefore the minimum separation between them and the causal event is set to 0. Note that an event can be considered *and_context* even if it is associated with an **or** expression. If the causal event of a rule with an **or** expression is its enabling event, then one other event is necessary in order for the **or** expression to be true when the rule becomes enabled. This event is *and_context* for the **or** rule. For events with **or** expressions, there is also an opposite restriction. Any events that would cause the value of the **or** expression to become true before the causal event fires must not be reordered to occur before the causal event. Therefore the maximum separation between e_i and e_{r_c} is set to 0 to ensure that e_{r_c} cannot happen after e_i . These entries in the POSET matrix ensure that none of the timing assignments allowed violate the reordering restrictions.

After the new constraints are added, the matrix is recanonicalized, which tightens all of the separations down to the maximum allowed by the known constraints. Finally, any events that are no longer relevant to future behavior of the system are removed from the matrix by the *project* function. An event can no longer effect future behavior if it is not causal to any rule currently in the constraint matrix and the direction of the signal transition no longer matches the current state ($a+$ no longer matches the current state if a is low in the current state). The result is a POSET matrix that constrains the minimum and maximum separations between events to bounds that are implied by the causality in the firing sequence. Once this new POSET matrix is computed, it is used to update the constraint matrix, as described in the previous chapter.

This algorithm extends the benefits of POSET timing to specifications with level expressions. The additions that are necessary to support levels do not add significantly to computation time, since they simply consist of determining causality and context relationships. When TEL structures are limited to simple **and** or **or** terms, these relationships can be determined by checks that occur when a rule becomes enabled, and require very little computation time.

5.5 Example

Figure 5.3 shows the application of the POSET algorithm applied to the TEL structure fragments at the top of the figure. Initially, assume that $a+$ and $v+$ fire at the same time and that the value of all signals other than $a+$ and $v+$ are false. Text between the matrices shows the currently firing event, the firing sequence, the causal event, and which events, if any, are *and_context* and *or_context*. Rule firings are not shown in these firing sequences since each event is enabled by a single rule, whose firing immediately precedes the event firing. Two versions of the POSET matrix are shown after each firing. The first shows the matrix after the algorithm sets the constraints but before it has been recanonicalized and extraneous events have been projected. The second column shows the matrices after they have been recanonicalized and projected. This distinction is made to show which separations are set explicitly by the algorithm and which separations are determined by the recanonicalization process.

The first event to fire after the initial events is $x+$. The rule enabling $x+$ has a level expression of **true**, so none of the extra conditions added by this chapter are used. The maximum separations between $x+$ and its causal event $v+$ is set to 5, the maximum

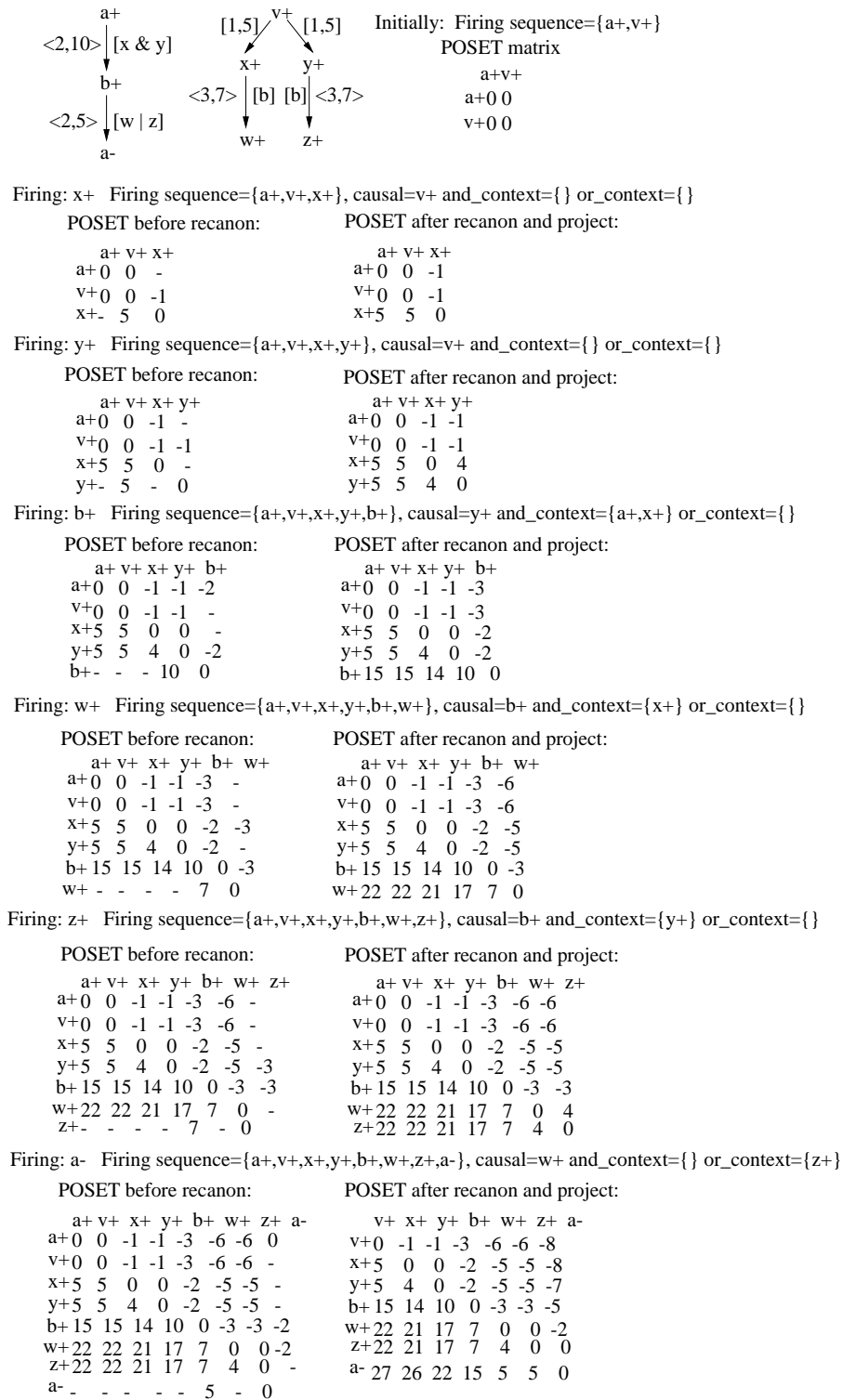


Figure 5.3. Example for POSET algorithm with levels.

bound on the rule connecting them. The minimum is set of -1, which is the minimum on $v+ \rightarrow x+$. The other separations are left unconstrained and set to infinity, as indicated by a dash (-). After the matrix is recanonicalized, the “-” entries are constrained down to the maximum allowed by the other entries. The recanonicalized matrix shows that $x+$ fires 1 to 5 time units after $a+$. Next, $y+$ fires. Its enabling rule also has no level expression, so the additional restrictions are not used. The resulting POSET matrix shows that $x+$ and $y+$ can fire in either order in the POSET defined by this firing sequence since $x+$ is allowed to fire up to 4 time units after $y+$ and $y+$ is allowed to fire up to 4 time units after $x+$. The firing of $y+$ causes the next firing, $b+$. This event is enabled by a rule with a boolean expression and is used to illustrate the additional reordering restrictions. Since $y+$ is causal to $b+$ the maximum separation between $y+$ and $b+$ is set to 10. There are also two *and_context* events, $a+$ and $x+$. These events must occur before $y+$ to ensure that it remains causal. The matrix indicates that the firing of $a+$ is already restricted to fire before $y+$ since $PM[a+][y+]$ is set to -1, but the separation between $x+$ and $y+$ needs to be restricted. The matrix that is created after the firing of $y+$ shows that $x+$ and $y+$ can fire in either order and the firing of $x+$ is allowed to occur up to 4 time units after the firing of $y+$. After $b+$ fires, the entry in the matrix $PM[x+][y+]$ is changed from 4 to 0, indicating that $x+$ is no longer allowed to fire after $y+$. Minimum separations are also added to the matrix between $y+$ and $b+$ since $y+$ is causal to $b+$, and $a+$ and $b+$ since $a+$ enables $b+$. After the matrix is recanonicalized, the unconstrained entries are filled. Notice that the minimum separation between $x+$ and $b+$ is 2, which is the minimum bound of the rule enabling $b+$. This occurs because $x+$ is not allowed to fire after $y+$ and $y+$ must occur 2 time units before $b+$. This illustrates how the restrictions added for context events ensure that they fire early enough. The next two event firings, $w+$ and $z+$ are similar to the firing of $b+$, they have a single literal in their level expressions and their enabling events are *and_context*. The final firing, $a-$, is different since its enabling rule has an **or** expression. Its causal event is $w+$. Its enabling event, $b+$, is *and_context* since it must fire before $w+$, and $z+$ is *or_context* since it must fire after $w+$. The entries in the matrix must be set so that this is the case. The minimum separation between $b+$ and $a-$ from the previous firing is already 3 so it does not need to be changed. After the previous firing $w+$ is allowed to fire after $z+$ since $PM[w+][z+]$ is 4. After the firing of $a-$, this separation is set to 0. Minimums are also set between the events $a-$ and $w+$, and the events $a-$ and $b-$. After

recanonicalization the unconstrained entries are filled in and entries concerning $a+$ are removed since it no longer matches the value of the signal a after the firing of $a-$.

5.6 Summary

The algorithm presented in this chapter allows for very expressive specifications to be analyzed using POSETS. Its description is quite complex, but the extensions add only minimal overhead to the algorithm. All of the computations necessary to extend the algorithm are done in constant time by storing relevant information when it is available. Since TEL structures with levels allow for circuits to be expressed more compactly, the extensions presented in this chapter significantly improve the performance of the algorithm.

CHAPTER 6

OPTIMIZATIONS

Show me a thoroughly satisfied man, and I will show you a failure.
- Thomas A. Edison

There are a number of optimizations to the POSET algorithm developed in the last two chapters that can reduce the number of geometric regions generated and decrease state space size. This chapter introduces five optimizations: subsets, supersets, untimed rules, merge, and interleaving. The first two optimizations, subsets and supersets, reduce the number of regions generated by checking if the current region contains or is contained in a region that has already been found. The untimed rule optimization reduces the number of regions generated by eliminating rule firing interleavings with rules that have a $[0, \infty]$ bound. The merge optimization reduces the number of untimed states found by considering markings equivalent if they would be equivalent in a Petri net. The interleaving optimization reduces state space size by eliminating some interleavings between rule firings from consideration. These optimizations significantly increase the size of examples that the algorithm can analyze.

6.1 Subsets

The simplest optimization is to check for *subsets* when checking to see if a region has been explored already. If a region is a subset of a region that has been explored, then all of its possible future behaviors are explored by the exploration of the larger region. Any exploration starting from the smaller region generates redundant regions. Checking for a subset can be done simply by checking to see if all of the entries in one matrix are smaller than their counterparts in the other matrix.

Some examples of region matrices are shown in Figure 6.1. The region in Figure 6.1(b) is a subset of the region in Figure 6.1(a) since all of the entries in Figure 6.1(b) are smaller than their corresponding entries in Figure 6.1(a). The matrices in Figure 6.1(c)

	r1 r2 r3		r1 r2 r3		r1 r2 r3		r1 r2 r3
r1	0	5	5	r1	0	5	4
r2	-1	0	0	r2	-1	0	0
r3	-1	0	0	r3	-2	0	0

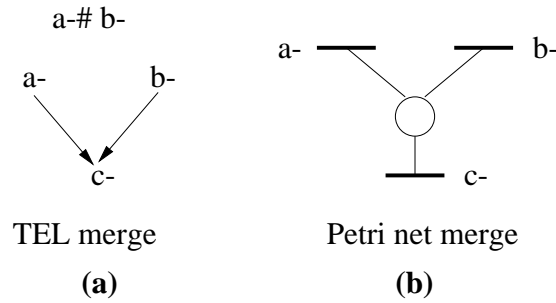


Figure 6.2. Merges.

6.3 Untimed Rule Optimization

If a rule has a timing bound of $[0, \infty]$ it is referred to as an untimed rule. Untimed rules enforce ordering between events but they do not specify any timing. An untimed rule is always satisfied as soon as it is enabled since its lower bound is 0, and it never restricts the firing times of other enabled rules since its upper bound is ∞ . Since untimed rules do not effect the firing times of other rules and are always satisfied as soon as they are enabled, they do not need to be included in the geometric region. When the untimed rule optimization is applied, entries for enabled untimed rules are not placed in the geometric region. The algorithm usually computes the list of rules that can fire by determining which rules represented in the current geometric region are satisfied. Therefore, this optimization requires that the algorithm also check the set of untimed rules to determine if any are enabled and add those that are to the list of rules that can fire. This optimization makes the regions smaller in specifications with many untimed rules and it can also make the state space smaller since regions which are generated by different interleavings of untimed rule firings are not distinguished.

6.4 Merge

The next optimization deals with merges, which are used in a specification to represent disjunctive **or** causality. Figure 6.2 shows examples of merges using a TEL structure and using a Petri net. In both cases, either the firing of $a-$ or $b-$ causes the firing of $c-$. Both specifications require that the firings of $a-$ and $b-$ are mutually exclusive. This is indicated explicitly in the TEL structure with the conflict and implicitly in the Petri net by the assumption that the net is one-safe.

When the algorithm is exploring the state space it compares the current set of enabled rules against the sets of enabled rules that have already been found and stored in the state table. If the sets of enabled rules are different, the algorithm assumes it has found a new state. State space exploration algorithms for Petri nets perform a similar operation. They check to see if the current marking has been found before, and if it has not they assume they have found a new state. Most of the time a rule in a TEL structure has a corresponding place in a Petri net. However, when the specification contains merges this is not the case. Figure 6.2(a) has two rules, while Figure 6.2(b) has only one place. An algorithm exploring the state space for a TEL structure finds a different set of enabled rules depending on whether $a-$ or $b-$ fired, but the algorithm exploring the Petri net finds the same marked place regardless of whether $a-$ or $b-$ fires. This may cause an algorithm which explores Petri nets to perform better than the POSET algorithm described in this thesis since the POSET algorithm has to find many more untimed states.

Fortunately, merges which correspond to Petri nets can be detected during state space exploration. If the current set of enabled rules differs from a set in the state space only by differences in a merge, the two sets of enabled rules can be considered equivalent. Figure 6.3 shows the algorithm for performing this check. It determines if the two rule sets it is given, R_1 and R_2 , are equivalent. When used in state space exploration, one of the rule sets is the current set of enabled rules and the other is a set of enabled rules in the state table. The algorithm tries to match every rule in R_1 to exactly one corresponding rule in R_2 . It first checks whether the rule sets are the same size. If they are not, then there is not a one-to-one match and the function returns false. Then, it checks to see if the rule $r_1 \in R_1$ is in R_2 . If it is, the algorithm moves on to the next rule. If r_1 is not in R_2 the algorithm checks if there are any rules in R_2 that share the enabled event of r_1 , f , and have enabling events that conflict with r_1 's enabling event, e . When the algorithm finds such an event, it sets *found* to true and continues to search all of the rules in R_2 . If it finds another match, then the merge is not a simple Petri net-like merge. It is a more complex structure that cannot be translated into a Petri net without creating additional places and transitions. When this kind of merge is detected, the algorithm returns **false**, indicating that the rule sets are different. If the algorithm searches through all of the rules in R_2 and cannot find a match, the the algorithm also returns **false** since there is no match to the rule from R_1 . If all of the rules in R_1 can be matched to exactly one rule in R_2 , the algorithm returns **true**. This algorithm does add

Algorithm 6.4.1 (Determine if two set of rules are equivalent)

```

bool merge_match(rule set  $R_1$ , rule set  $R_2$ ){
  if( $|R_1| \neq |R_2|$ ) return false;
  forall ( $r_1 = \langle e, f, l, u, b \rangle \in R_1$ ){
    if ( $r_1 \notin R_2$ ) then
      bool found = false;
      forall ( $\langle e', f, l, u, b \rangle \in R_2$ ){
        if ( $e \# e' \wedge \neg found$ ) then
          found = true;
        else if ( $e \# e' \wedge found$ ) return false
      }
      if ( $found = false$ ) return false;
    }
  }
  return true;
}

```

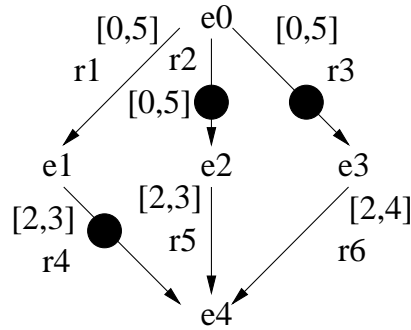
Figure 6.3. Procedure for matching two rule sets.

some a slight amount of overhead ($< 1\%$) to the algorithm, but for specifications with many merges, it significantly reduces memory consumption and runtime.

In order to get the full benefit of this optimization, regions need to be checked for merge equivalence as well. Without this optimization, regions are only considered equivalent if they represent age differences between the same set of enabled rules. However, with the optimization, if two sets of enabled rules are determined to be equivalent by the matching algorithm, then rules in the region can be matched as well. If region M_1 contains rule $r_1 = \langle e, f, l, u, b \rangle$ and region M_2 contains rule $r_2 = \langle e', f, l, u, b \rangle$ and $e \# e'$, then M_1 can be compared to M_2 by substituting r_1 for r_2 when doing the age comparisons. This optimization prevents state space exploration using TEL structures from performing worse than state space exploration using Petri nets on specifications with many merges.

6.5 Interleaving

The previously described optimizations change the way the algorithm determines if two states are the same. This optimization reduces the state space by preventing certain redundant timed states from being generated by removing certain interleavings between rule firings from consideration. The purpose of exploring different interleavings between rule firings is to ensure that all possible causal rules for each event firing are explored.



Firing sequence: e_0, r_1, e_1

Figure 6.4. Example of interleaving optimization.

If two different rule firing interleavings result in the same causal rule for a given event firing, no additional information is generated by exploring both of them, due to the way the POSET algorithm generates POSET matrices. When information on a new event, e , is added to the POSET matrix, the causal rule determines the upper bound on the time separation between the firing of e and its causal event. If the causal rule has no level expression, two firing sequences with the same causal rule for e always result in the same time separations between the firing of e and the other events in the matrix.

Consider for example, the TEL structure in Figure 6.4. Initially, the firing sequence r_1, e_1 has been explored. Since there are many possible interleavings between the firing of r_4 and the firing of the other rules in the TEL structure, it reduces execution time if only one interleaving where r_4 is causal to e_4 is explored. Figure 6.5 shows the POSET matrices generated as e_4 fires when each of the rules enabling e_4 is causal. The POSET matrices show that when r_4 is causal to e_4 , it generates a unique matrix that is not a subset of the matrices generated when the other rules are causal. The matrix in Figure 6.5(a) is the matrix generated whenever r_4 fires last, regardless of whether r_4 is enabled first or last. For example, the firing sequences $r_1, e_1, r_2, e_2, r_3, e_3, r_6, r_5, r_4, e_4$ and $r_3, e_3, r_2, e_2, r_1, e_1, r_5, r_6, r_4, e_4$ result in the generation of the same POSET matrix. Since there are multiple firing sequences where r_4 fires last, this POSET matrix is generated multiple times when all rule firing interleavings are explored. Additionally, since a different geometric region is generated for each rule firing interleaving, many additional geometric regions are generated by exploring all of the rule firing interleavings which are

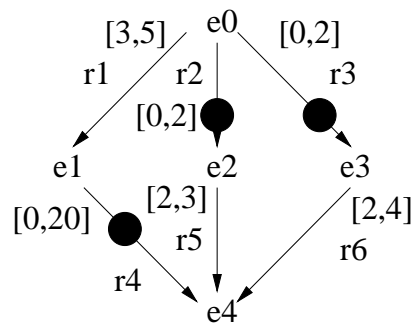
POSET matrix: r4 is causal					POSET matrix: r5 is causal					POSET matrix: r6 is causal				
	e1	e2	e3	e4		e1	e2	e3	e4		e1	e2	e3	e4
e1	0	5	5	-2	e1	0	5	5	-2	e1	0	5	5	-2
e2	5	0	5	-2	e2	5	0	5	-2	e2	5	0	5	-2
e3	5	5	0	-2	e3	5	5	0	-2	e3	5	5	0	-2
e4	3	8	8	0	e4	8	3	8	0	e4	9	9	4	0
	(a)					(b)					(c)			

Figure 6.5. POSET matrices with various causal places.

going to create the same POSET matrix. In order to reduce the number of interleavings explored, the algorithm should only generate the POSET matrix in Figure 6.5(a) once, and not explore the other rule firing interleavings that lead to it.

The difficulty is deciding when a rule can be fired without interleaving it with other token firings, and when it must be interleaved so it has a chance to be causal. In general, solving this problem could involve examining the entire firing sequence that has been explored so far. However, in certain cases, interleavings can be eliminated by a structural examination of the TEL structure. It is difficult to do this analysis for events which are enabled by rules with non-empty choice sets or boolean expressions, therefore, if a rule r , enables an event e which is enabled by a rule with a non-empty choice set, firings of r are always interleaved. Also, if r has a boolean expression, r is always interleaved.

The goal of the optimization is to generate only one POSET matrix per causal rule. One way to do this is to stipulate that a rule can only fire last if it is enabled last. If a rule, r , is enabled while other rules that enable r 's enabled event are not enabled, then it is always fired as soon as a region is created that allows it to meet its lower bound. Firing sequences where it fires later than this are not considered. If all of the rules that enable an event are enabled last in some firing sequence, then this produces exactly one firing sequence where each rule is causal. This is the case in the example in Figure 6.4, since each token can be created last in some firing sequence. If this method is used on the example in Figure 6.4, firing sequences where r_4 is causal and receives its token last such as: $r_3, e_3, r_2, e_2, r_1, e_1, r_5, r_6, r_4, e_4$, and $r_3, e_3, r_2, e_2, r_1, e_1, r_6, r_5, r_4, e_4$, are explored. The firing sequence $r_1, e_1, r_2, e_2, r_3, e_3, r_5, e_6, r_4, e_4$ and any other firing sequence where r_4 is causal and does not receive its token last are eliminated. When rule r_4 is enabled and



Firing sequence: e_0, r_1, e_1

Figure 6.6. TEL structure where p_4 cannot be created last.

Restricted POSET matrix:
 r_4 is causal and e_1 fires first

	e_1	e_2	e_3	e_4
e_1	0	5	5	-2
e_2	5	0	5	-2
e_3	5	5	0	-2
e_4	3	3	3	0

Figure 6.7. A restricted POSET matrix.

other rules which enable e_4 are not, the rule r_4 is fired as soon as c_4 can reach age 2, its lower bound. This approach produces a correct result for this example. However, if the timing bounds are changed to those in Figure 6.6, r_4 cannot be enabled last, but it can fire last. Clearly, in this case, a part of the state space is eliminated if the firing of r_4 is not interleaved and a firing sequence where r_4 is causal is never explored. This indicates that there are some cases where the optimization cannot be made.

In order to make the optimization in the algorithm, it is necessary to determine under what circumstances a rule can be enabled earlier than the other rules and still add new behavior when it is causal. POSET matrices contain timing assignments for all of the valid reorderings of the firing sequence being explored. The set of timing assignments allowed by a particular set of reorderings can be found by restricting the POSET matrix

in a way that forces events to fire in the desired order. This illustrates which firing sequences are responsible for which timing assignments allowed by the POSET matrix. Figure 6.7 shows a restricted POSET matrix for the TEL structure from Figure 6.4. The rule r_4 is causal to e_4 and the matrix is restricted so that e_1 must fire before e_2 and e_3 , by placing zeroes in the first row. This restriction implies that only firing sequences where r_4 is enabled first are represented. This POSET is a subset of all of the POSETs shown in Figure 6.5, which indicates that the set of timing assignments allowed when r_4 is enabled first and is causal are also allowed when any of the other rules are causal and when r_4 is enabled later and is causal. Therefore, it is not necessary to explore the firing sequence where r_4 is enabled first and is causal. The firing of r_4 does not need to be interleaved when it is enabled first because the upper bound on r_4 is less than or equal to the upper bounds on the other two rules that enable e_4 in Figure 6.4. In Figure 6.6, r_4 does need to be interleaved since its upper bound is greater than the upper bound of the other rules that enable e_4 .

To generalize, consider a rule r with bounds $[l, u]$, which enabled event e . The event e is enabled by a set of rules $\{r_1, \dots, r_n\}$, all of whose upper bounds are greater than or equal to u . Now, assume that r is enabled at time δ , and r is causal to e in the current firing sequence. This means that event e fires no later than $\delta + u$. All of the the rules that enable e and are not enabled when r becomes enabled are enabled either at time δ or some time later. Since e fires no later than $\delta + u$, the maximum clock age for a rule that is not enabled when r becomes enabled is u or less. All $r_i \in \{r_1 \dots r_n\}$ have maximum bounds which are greater than u , therefore their clocks cannot exceed their upper bounds when r is causal and becomes enabled earlier. Any of the rules that are not enabled when r becomes enabled can be fired after r without exceeding their upper bounds. If any of these rules fires after r , then r is not causal. Therefore, the clocks for rules that are not enabled when r becomes enabled do not reach any values that are not guaranteed to be allowed by a firing sequence where r is not causal. Rules that are enabled when r becomes enabled may have clocks that exceed their upper bounds. However, these clocks have even larger values in sequences where a rule that is not enabled is causal to e . This means that as the algorithm is exploring a firing sequence σ where r has become enabled before the other rules that enable e , any region that is generated by a continuation of σ where r is causal to e is a subset of a region created by a firing sequence where r is not causal to e . Thus, the algorithm does not need to explore continuations of σ where

r is causal to e , and it does not need to interleave the firing of r with all the other rule firings to ensure that it has a chance to be causal. In specifications containing events that are enabled by a large number of rules, this optimization can produce a significant reduction in runtime. It also produces a significant reduction in the number of geometric regions generated since a new region is generated for every rule firing interleaving, and the optimization reduces the number of rule firing interleavings.

6.6 Summary

A large number of the regions that are generated by the unoptimized POSET algorithm are redundant. These redundant regions significantly reduce the size of the specification that can be analyzed. The optimizations presented in this chapter remove most of these redundant regions and result in significant performance improvement. The next chapter describes how the regions can be stored more compactly to save memory.

CHAPTER 7

IMPLICIT METHODS

*The biggest difference between time and space
is that you can't reuse time.*
- Merrick Furst

Memory is often the limiting factor when attempting to synthesize or verify a timed system. Even though the POSET algorithm dramatically reduces the number of regions generated, the algorithm still requires a great deal of memory for large, complex specifications. The optimizations discussed in the previous chapter address this problem by reducing the number of regions generated, reducing the size of the regions generated, or reducing the size of the stack. To further reduce the amount of memory needed, implicit methods can be used to more efficiently represent the state space. This chapter describes a method for representing geometric regions using implicit methods, first presented by Thacker in [66, 65], which significantly increases the size of specification that can be analyzed.

7.1 Representing Geometric Regions

Much of the data compiled during state space exploration consists of bit vectors. Therefore, Bryant's binary decision diagrams (BDDs), which are a highly efficient method for storing and manipulating Boolean functions[16] are used to represent the untimed states. Geometric region information is integer-valued and standard BDDs can only represent binary data. Therefore multi-terminal binary decision diagrams MTBDDs are used to store the region matrices. MTBDDs are a type of BDD which allow terminal nodes to contain 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, when BDDs are used, 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 first step in building an implicit representation is to use BDDs to store the bit vector that indicates which rules are in R_m (the set of rules whose enabling events have fired). To accomplish this, an atomic BDD is allocated to represent each rule. These BDDs are assembled into the array $\mathbf{m} = (m_1, \dots, m_n)$, where n is the number of rules in the TEL structure. An atomic BDD is one which represents a single variable. As shown in Algorithm 7.1.1, 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 a TEL 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 7.2(a). (Note that BDDs as shown are drawn to be relatively readable, and do not necessarily indicate the actual node ordering or machine representation of these structures.)

Algorithm 7.1.1 (Extract R_m BDD)
bdd FindR_mBDD(rule set R, rule set R_m, bdd array m) {
 bdd $\beta = TRUE$
 foreach ($r_i \in R$)
 if ($r_i \in R_{en}$) **then**
 $\beta = \beta \wedge \mathbf{m}[i]$
 else
 $\beta = \beta \wedge \neg \mathbf{m}[i]$
 return β
}

Figure 7.1. Function to extract a BDD for the rule set R_m .

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 i^{th} matrix associated with a given R_m set. Any number i can be viewed as a bit vector $\vec{i} = (i_0, \dots, i_n)$, where i_0 is the low order bit of the binary representation of i , and i_n is

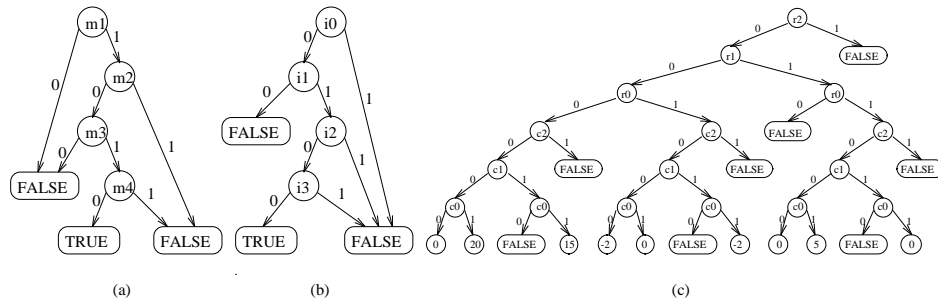


Figure 7.2. MTBDD representation of (a) R_m , (b) the number “2”, and (c) a geometric region matrix.

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 7.2(b) represents the number “2” in a four-bit notation. Numbers of this form are used to create a dynamically sized array of matrices. In order to conserve space, precisely enough bits are used to represent the largest number currently needed.

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 $\vec{r} = (r_0, r_1, r_2, \dots, r_n)$ or $\vec{c} = (c_0, c_1, c_2, \dots, c_n)$, so the function can be viewed as $M(\vec{r}, \vec{c}) = M_{rc}$. MTBDDs presented by Clarke in [24] are an ideal way to represent this type of function. BDDs are constructed for each necessary row and column index, and stored in arrays \mathbf{r} and \mathbf{c} . The BDD for the i^{th} column index is stored in $\mathbf{c}[i]$ and the BDD for the i^{th} row index is stored in $\mathbf{r}[i]$. For example, $\mathbf{r}[3]$ 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 7.3 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 7.2(c) 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 7.1.2 (Construct Matrix MTBDD)

```

mtbdd MakeMatrixBDD(int n, matrix M, bdd vector r, bdd vector c) {
  mtbdd  $\beta = FALSE$ 
  forall (i :  $0 \leq i \leq n$ )
    forall (j :  $0 \leq j \leq n$ )
      {
        if  $M[i, j] \neq \text{“not\_an\_entry”}$  then
          {
            bdd  $\alpha = \mathbf{r}[i] \wedge \mathbf{c}[j]$ 
            mtbdd  $\gamma = \text{terminal}(M[i, j])$ 
             $\beta = ITE(\alpha, \gamma, \beta)$ 
          }
        }
      }
  return  $\beta$ 
}

```

Figure 7.3. Function to create a MTBDD for the matrix M .

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 7.4 shows the complete MTBDD for the timed state where $R_m = \{r_1, r_3\}$, the link value is 2, and

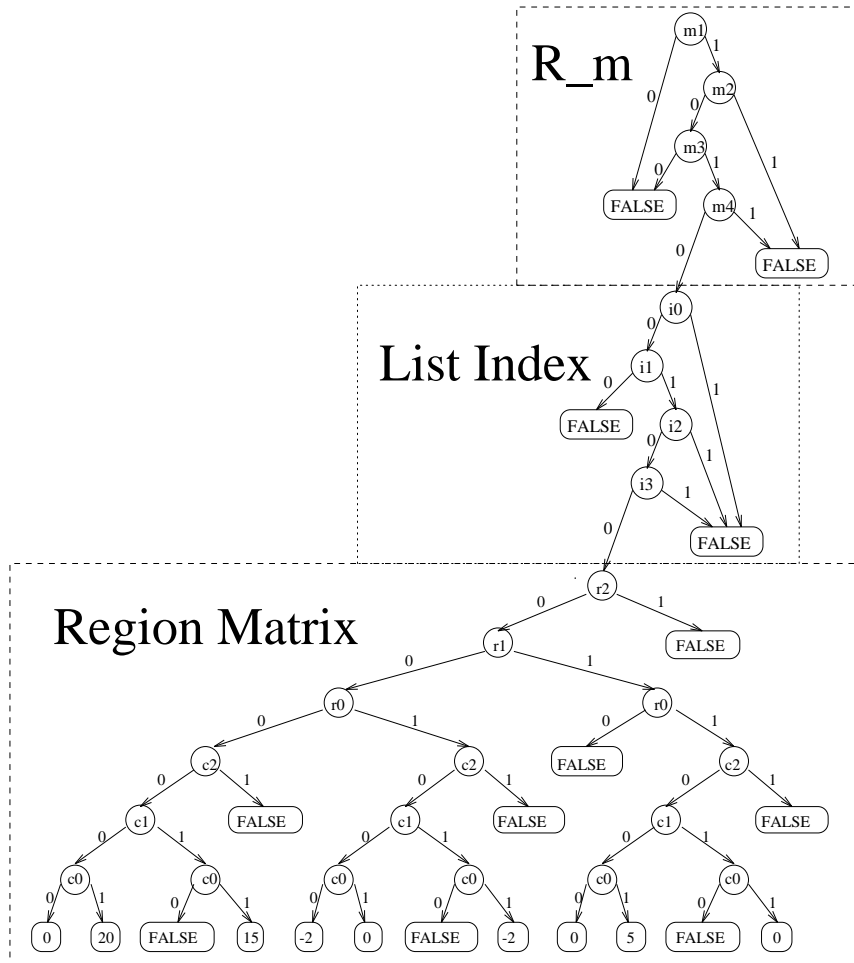


Figure 7.4. MTBDD representation of a timed state.

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 7.5, 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 7.5(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 7.5(b). Note that adding a third matrix (as shown in Figure 7.5(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.

7.2 Representing the Reduced State Graph

The goal of state space exploration for synthesis is to find all of the boolean states of the system and the possible transitions between them. This information is necessary in order for asynchronous logic synthesis algorithms to generate a circuit from the state space, and it must be stored in addition to the region MTBDDs during state space exploration. This set of reachable states and transitions is referred to as the *reduced state graph*, or RSG. To store the RSG, a pair of BDDs, Φ and Γ , are constructed as the space is explored. The BDD Φ is the characteristic function representation of the reachable untimed states. The state vector $\vec{s} = (s_0, s_1, \dots)$ represents the binary values of the signals in a given state. 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. As each new untimed state is found, a BDD s , which represents the current signal values, is constructed in a fashion similar to that used for the R_m set. Φ , which represents the total state space, is then extended by taking the logical OR of the current state and the current value of Φ ($\Phi = \Phi \vee s$). For example, the RSG shown in Figure 7.6(a) shows the reachable state space for a circuit with three signals, a , b , and c . Figure 7.6(b) shows the BDD for the initial state $RR0$,

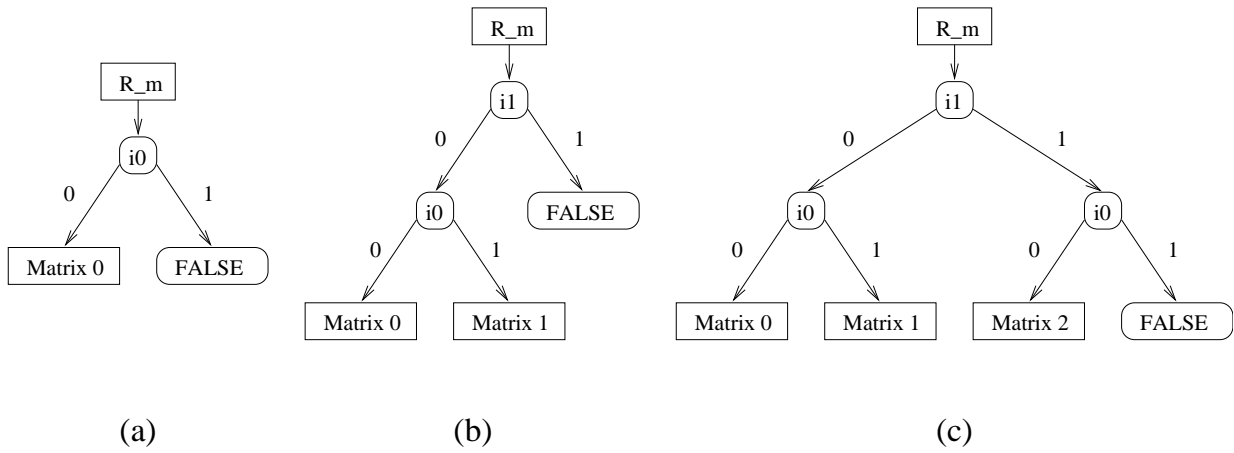


Figure 7.5. A false-terminated array holding (a) one, (b) two, or (c) three matrices.

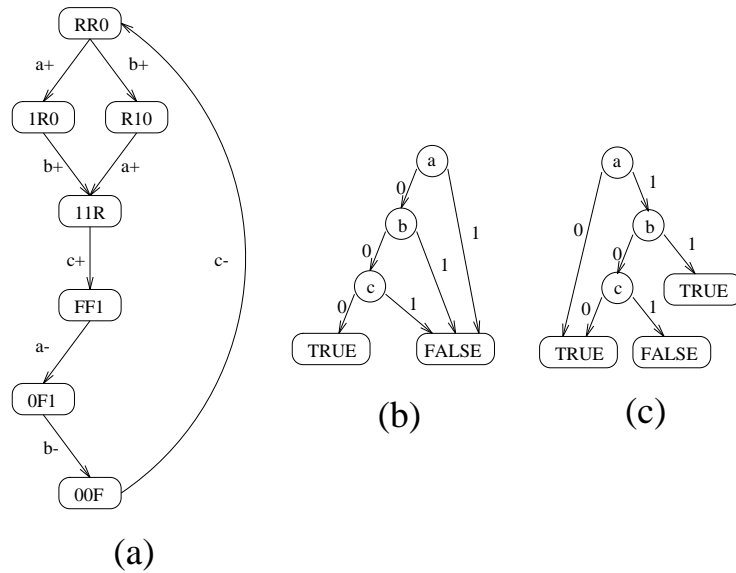


Figure 7.6. (a) A reduced state graph (RSG), (b) a BDD for the state RR0, and (c) the characteristic function BDD (S) for the state space.

and Figure 7.6(c) shows the characteristic function BDD Φ for this state space. All states are reachable except for $F01$. Γ is the characteristic function representation of the transition relation and is constructed in an analogous manner. Each pair of states (s, s') is represented by a pair of vectors \vec{s} and \vec{s}' , which indicate the values of each signal in the two adjacent states joined by a transition. A complication arises from the use of timing information in the exploration of the state space. When the timing analysis shows a state to be unreachable, it is not included from the state space. If these states are ignored the signal enablings leading to each of them would be lost. Because timed circuit synthesis is highly dependent on this information, circuits derived from such a state space would be suboptimal and possibly incorrect. To prevent this situation, a transition is inserted into N for every enabled signal, even if this is a “ghost” transition leading to a timed unreachable state. Construction of an implicitly represented reduced state graph in this way, not only reduces memory consumption, but also allows implicit methods to be applied to logic synthesis [66].

7.3 Summary

Representing the timed state space using implicit methods typically produces very significant memory savings when applied to large examples. However, since the core of the algorithm still operates on explicit matrix representations it requires a translation from an explicit matrix to an MTBDD every time a state is stored in the state table. This produces a large amount of runtime overhead and the BDD optimization degrades runtime. It is most useful for large examples which run out of memory using the explicit approach.

CHAPTER 8

VERIFICATION

*Testing can prove the presence of bugs, but
never their absence.
-Edsger Dijkstra*

Timing verification is essential in order to successfully design a timed system. Even when timing information is designed in from the beginning, it is necessary to verify that the physical implementation meets the requirements of the specification. State space exploration is the core problem in timed system verification, which makes the POSET algorithm directly applicable to verification. However, one element is missing. The TEL structure specification language defined in Chapter 2 does not provide a way to define properties to be verified. This chapter presents a method for property specification, and formally defines the set of sequences for which verification succeeds and fails.

Many logics have been developed to specify temporal behavior of untimed concurrent systems, such as LTL presented by Pnueli [56] and CTL presented by Browne [14], Dill [30] and Clarke [23]. The logics for untimed concurrent systems do not deal with concrete time values. For example: a occurs implies that eventually b occurs can be specified, but a occurs implies that b occurs before 10 time units cannot be specified. Since these logics do not deal with concrete timing bounds, any properties that can be specified using them can be checked after the timed state space is found by examining the reduced state graph. This graph contains all reachable boolean states and all possible transitions between them, and therefore completely describes the untimed behavior of the system. Although ATACS does not currently contain a facility to check LTL or CTL formulas against the state graph, Burch presents an algorithms exist to do so in [17].

The problem of specifying and checking concrete timing bounds is more complex. Concrete timing requirements cannot be checked by an examination of the reduced state graph since it does not contain concrete timing information. Concrete timing requirements could be checked by examining the state graph along with the regions

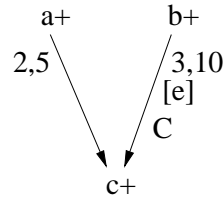


Figure 8.1. Example of a constraint rule

generated for each state, but the information is more accessible during state space exploration. Methods have been proposed for the specification of verification properties with concrete timing bounds including complex and expressive temporal logics such as Alur’s TCTL [2], Yoneda’s TNL [73], Alur’s MITL [4], and Alur’s TPTL [5]. The POSET algorithm could be used to verify properties expressed using these logics. However, it would require significant modification and would add complexity and overhead to the process. Additionally, one the of goals of this thesis is to produce a specification method that is accessible to circuit designers. Logics such as TCTL, TNL, MITL, and TPTL require a significant amount of mathematical understanding in order to use them effectively. Therefore, we choose an approach that is simpler and less expressive, but easier to understand.

8.1 Constraint Rules

Verification properties are specified using a set of *constraint* rules: $C \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b : \{0, 1\}^N \rightarrow \{0, 1\})$. These constraint rules are similar to the constraint places described by Rokicki in [58]. Constraint rules never actually fire and they never appear in a firing sequence. Instead, the constraint rules are checked each time a rule or event is added to the firing sequence. Failures caused by constraint rules arise due to two conditions:

1. An event fires and any constraint rule enabling it is not satisfied.
2. A firing results in a sequence which can be given a timing assignment which causes the clock on a constraint rule to exceed its upper bound.

Figure 8.1 shows a TEL structure fragment which contains a constraint rule. This rule requires that the TEL structure must meet a number of requirements. The first require-

ment is that $c+$ must fire no more than 10 time units after the rule $\langle b+, c+, 3, 10, [e] \rangle$ becomes enabled. If $c+$ can ever fire later than this, the age of the constraint rule exceeds its upper bound and causes a failure. The next requirement is that $b+$ must fire at least 3 time units before $c+$ fires, and the signal e must be high at least 3 time units before $c+$ fires. These conditions are necessary in order for the constraint rule to be satisfied when $c+$ fires. If the constraint rule were disabling, then the rule would also require that e must remain high from the time it rises to the time that $c+$ fires. This single constraint rule specifies a rather complex set of behavior requirements. Constraint rules, especially when combined with the ability to specify sequencing events, provide a reasonably powerful way in which to describe the behavior to be verified.

8.2 Success and Failure Sequences

In [27, 29], Dill presents a method for verifying speed independent circuits using *trace theory*. This approach is based on dividing the set of possible execution sequences allowed by the specification into a failure set, F , and a success set, S . The method is extended to work with timed specifications by Burch in [18, 19]. In this method, explicit time advancement events are included in the specification and in the execution sequences. This allows concrete time properties to be verified with trace theory. This section adapts the trace theory concepts to firing sequences and the continuous time model that is used throughout this thesis.

In order to describe how trace theory can be applied to firing sequences, we must first formally define the behavior of constraint rules. The first definition concerns constraint rule enablings. Although constraint rules never fire, they are enabled in the same way as standard rules. A constraint rule $r \in C$ is enabled in $\sigma_{0..n}$ if it satisfies Definition 2.2.3. Since constraint rules do not appear in the firing sequence, and therefore cannot be given an explicit timing assignment, the definitions concerning their timing behavior differ from the definitions in Chapter 2 which concern standard rules.

Definition 8.2.1 A rule $r = \langle e, f, l, u, b \rangle \in \text{constraint_satisfied}(\tau, \sigma_{0..n})$ iff:
 $r \in C \wedge r \in \text{enabled}(\sigma_{0..n}) \wedge (\tau(\sigma_n) - \tau(E_m(r, \sigma_{0..n}))) \geq l$

This definition states that the constraint is satisfied by firing sequence $\sigma_{0..n}$ and timing assignment τ if τ causes the constraint rule to become enabled at least l time units before the last firing in the sequence. The next definition deals with the upper bound on the

constraint rule.

Definition 8.2.2 A rule $r = \langle e, f, l, u, b \rangle \in \text{constraint_expired}(\tau, \sigma_{0..n})$ iff:
 $r \in C \wedge r \in \text{enabled}(\sigma_{0..n}) \wedge (\tau(\sigma_n) - \tau(E_m(r, \sigma)) + \text{max_advance}(\sigma_{0..n}, \tau) > u)$

This definition states that the constraint is exceeded by firing sequence σ and timing assignment τ if τ allows the age of r to exceed u before another rule or event is forced to fire. These definitions are sufficient to define the a set of failure sequences.

There are are four conditions that place a firing sequence in the failure set, F . They are first described informally, and then presented formally below. The first condition that causes a failure is that the age of any clock associated with a constraint rule exceeds its upper bound. This indicates that an event has not fired soon enough. The second condition is that some constraint rule with e as an enabled event is not satisfied when e fires. If this condition occurs, an event is firing too early. The third condition is that a disabling rule becomes enabled and then loses its enabling. This indicates a hazard in the circuit. The final condition is that a finite firing sequence is generated from a cyclic specification. This indicates a deadlock.

We can now formalize the definition of the failure set, F . The null firing sequence, containing no firings, is in S . Therefore, there is a prefix of every sequence that is not in F . The failure set can be described by defining which firings, when added to a sequence $\sigma \in S$ cause the resulting sequence and all extentions of the resulting sequence to be in F .

Definition 8.2.3 Assuming that $\sigma_{0..n} \in S$, $\sigma_{0..n+1} \in F$ and $\sigma_{0..n+1}(E|R)^* \in F$ iff one of the following conditions holds:

1. $\exists \tau \in \text{valid}(\sigma_{0..n+1}) : \text{constraint_expired}(\tau, \sigma_{0..n+1}) \neq \emptyset$.
2. $L(\sigma_{n+1}) \in E \wedge \exists \tau \in \text{valid}(\sigma_{0..n+1}), r = \langle e, L(\sigma_{n+1}), l, u, b \rangle \in R :$
 $r \notin \text{constraint_satisfied}(\tau, \sigma_{0..n+1})$.
3. $\exists r = \langle e, f, l, u, b \rangle \in R : r \text{ is disabling} \wedge r \in \text{enabled}(\sigma_{0..n}) \wedge \neg b(\phi(\sigma_{0..n+1}))$.
4. $\text{firable}(\sigma_{0..n+1}) = \emptyset \wedge \text{enabled}(\sigma_{0..n+1}) = \emptyset$.

The first condition states that the addition of any rule or event firing which creates a sequence which has a valid timing assignment where a constraint rule exceeds its upper bound is a failure. The second condition is that an event firing causes a failure if any of

Algorithm 8.3.1 (Check deadlock)
 $RL = \text{find_timed_enabled}(TS, TEL, M);$
if ($RL = \emptyset$) **return fail**

Figure 8.2. Check for deadlock.

its enabling rules are not satisfied when it fires. The third condition is that any firing which causes the boolean expression on an enabled disabling rule to become false causes a failure. The final condition is that the enabled and firable sets are empty. This indicates that the sequence ends at σ_{n+1} and is finite.

These conditions completely describe the failure set F . The success set, S contains all sequences in Σ which are not in F ($S = \Sigma - F$). The state space algorithm can now be modified to generate a failure whenever a firing causes the generation of a failure sequence.

8.3 Checking for Failures

The state space exploration algorithm presented in Chapter 3 already checks for one type of failure, the disabling of a rule when its boolean expression becomes false. It now needs to check for three more conditions: a deadlock, a rule exceeding its upper bound, and an event firing with an unsatisfied constraint rule.

Deadlock is the simplest check. The algorithm assumes that the specifications it is given are cyclic and that a deadlock is always a verification failure. The modification to the state space exploration algorithm from Figure 3.1 is shown in Figure 8.2. Every time the algorithm generates a new timed state, it computes the set of rules that are allowed to fire in that state and places them in RL . If the RL set is ever empty, then no rules can fire from this state. If a rule cannot fire, an event cannot fire either, since the firing of an event requires the firing of a rule. Therefore, if RL is empty, the current sequence has satisfied condition 4 of Definition 8.2.3 and the algorithm generates a **fail** result.

The modification of the algorithm to check timing violations on constraint rules is more extensive. Since constraint rules do not fire, they are handled differently than standard rules. Clocks are created for them, but these clocks cannot be allowed to constrain the firing times of other rules. The first modification to the algorithm concerns the *find_timed_enabled* function from Figure 3.2. This function is modified to prevent constraint rules from being added to the rule list. The modification, shown in Figure 8.3,

Algorithm 8.3.2 (Find timed enabled)
rule_list RL find_timed_enabled($R_{en}, M, TEL\langle N, s_0, A, E, R, R_0, \#, C \rangle$) {
 for each ($r = \langle e, f, l, u, b \rangle \in R_{en}$) {
 if ($min_clock_value(r, M) \geq l \wedge r \notin C$) *add_list*(r, RL);
 return *RL*;
}

Figure 8.3. Find timed enabled rules.

simply checks if a rule is a constraint rule before adding it to the rule list. This prevents constraint rules from firing. The next modification is to the *update* function. It ensures that constraint rules do not restrict the firing times of normal rules, and checks that the timing bounds on constraint rules are always satisfied. The new version of the update function for the POSET algorithm is shown in Figure 8.4. First, the function updates the POSET matrix if an event fires. Then it projects the firing rule. Next it checks to see if any constraint rules have exceeded their upper bounds. If a rule has exceeded its upper bound, the algorithm generates a failure. It then advances time by setting the maximum age of all normal rules to the maximum bound on the rule and the maximum age of all constraint rules to infinity. Setting the maximum for constraint rules to infinity ensures that these bounds do not constrain the size of the region. The matrix is then recanonicalized. The next step is to check whether the minimum ages for all constraint rules which enable the firing event are met. After the minimum age is checked, each constraint rule which enables the firing event is projected

8.4 Example

Figure 8.5 illustrates the behavior of the new update algorithm applied to the TEL structure fragment at the top of the figure. Clearly, both the lower and upper bound on the constraint rule in the figure are violated. The figure shows how the algorithm determines this. Initially, the POSET contains only a , and the constraint matrix contains ages for rules $\langle a, b \rangle$ and $\langle a, c \rangle$. Since both are enabled by a , their age difference is 0. Since the maximum age in the matrix for the constraint rule, is 5, in this state it does not violate its upper bound. The minimum age on the constraint rule, 0, does violate its lower bound, 5, but this does not generate a failure since the event it enables, c , cannot fire yet. There is still time for $\langle a, c \rangle$ to reach its lower bound. When b fires, the POSET is updated to show that b fires between 2 and 5 time units after a . The constraint

Algorithm 8.3.3 (Update)
 void update(*TEL structure TEL* $\langle N, s_0, A, E, R, R_0, \#, C \rangle$, *geometric region M,*)
 POSET matrix PM, rule $r = \langle e, f, l, u, b \rangle$, *rule set* R_{en} , *bool event_fired*) {
 if(*event_fired*) **then**
 update_POSET(*TEL, PM, M, r, R_{en}*);
 project(*M, index(r)*);
 forall($r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in C$) {
 if ($r_i \in R_{en} \wedge M[0][index(r_i)] > u_i$) **then return fail**;
 forall($r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_{en}$)
 if($r_i \in C$) **then** $M[0][index(r_i)] = \infty$;
 else $M[0][index(r_i)] = u_i$;
 }
 reanoncalize(*M*);
 normalize(*M*);
 if(*event_fired*) **then** {
 forall($r_i = \langle e_i, f, l_i, u_i, b_i \rangle \in C$) {
 if ($r_i \notin R_{en} \vee M[index(r_i)][0] > -l_i$) **then return fail**;
 project(*M, index(r_i)*);
 }
 }
 }

Figure 8.4. Update the region and check constraints.

matrix now contains the constraint rule $\langle a, c \rangle$ and the normal rule $\langle b, c \rangle$. When time is advanced, the maximum age of $\langle b, c \rangle$ is set to 5, its upper bound. The maximum age for the constraint rule is set to infinity so that its upper bound does not constrain the matrix. When the matrix is recanonicalized the constraint rule has a maximum age of 10, and a minimum age of 2. The maximum exceeds its upper bound. The algorithm detects this and generates a failure. However, in order to illustrate how lower bounds are checked, suppose that the algorithm continues. When c fires, the maximum age of $\langle a, c \rangle$ is now 15 which still violates the upper bound constraint. The rule also now violates its minimum constraint as well. Its enabled event, c has fired, and its minimum age, 4, does not reach its minimum bound, 5. Therefore, the algorithm generates a failure here for a minimum violation. If the algorithm were to continue, the rule $\langle a, c \rangle$ would be projected from the matrix here since its enabled event has fired.

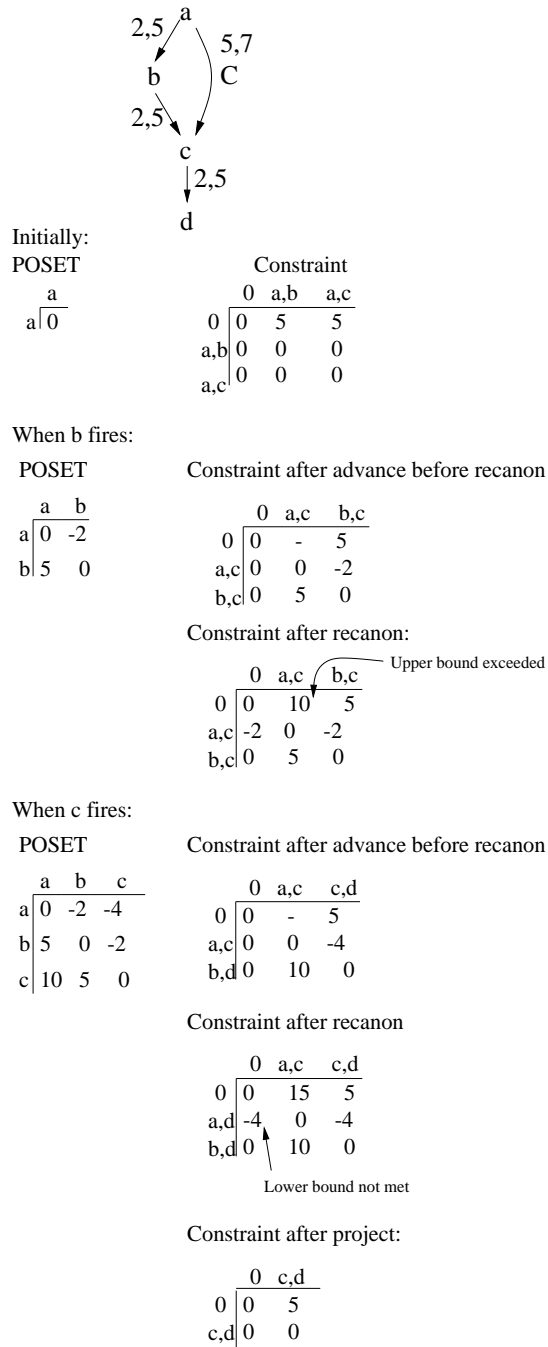


Figure 8.5. Example of new update algorithm.

8.5 Summary

The algorithm does not change significantly when constraint rules are added and checked. Their maximum ages are set to infinity instead of their upper bounds, and they must be projected when their enabling event fires instead of being individually fired and projected. The constraint checks require a simple examination of matrix entries. Constraint rules do add some overhead to the state space exploration process because their ages must be computed in addition to the ages of the normal rules. However, they add no algorithmic overhead and allow for the specification and verification of real time constraints in a way that is easily understood and used by circuit designers.

CHAPTER 9

RESULTS

*In theory, there is no difference between theory
and practice. But, in practice, there is.
-Jan L.A. van de Snepscheut*

The POSET algorithm dramatically reduces the number of geometric regions generated during state space exploration of highly concurrent systems. The new algorithm, along with the optimizations discussed in the previous chapters, is implemented within the CAD tool **ATACS** and produces very good results as illustrated by the examples in this chapter. Although some examples in this chapter are discussed in the context of synthesis and others in the context of verification, the same implementation of the POSET algorithm is used to find the state space of both types of examples.

The first set of results compares the POSET algorithm to **Orbits**[58] and demonstrates that the ability to directly analyze specifications with multiple behavioral rules results in a large performance improvement. The second set of results compares the POSET algorithm to timing approaches that are not based on geometric regions. These results show that the POSET algorithm makes the geometric region representation competitive when delay ranges are small and superior when they are large. The third set of results presents the impact of the BDD method on runtime and memory usage. It shows that using BDDs often produces an order of magnitude reduction in memory requirements, but also can severely impact runtime. The final set of results describes the application of the POSET algorithm and TEL structures to real world synchronous circuits from the IBM **guTS** microprocessor. Two of the examples, are analyzed using both a purely event based specification and a mixed level and event based specification. Results on these examples indicate that the more concise, level based representation produces a performance improvement. We believe that these results show that the algorithms developed in this thesis are not only useful for the analysis of asynchronous circuits, but also for any high performance circuit where aggressive timing assumptions are made.

9.1 Comparison with Orbits

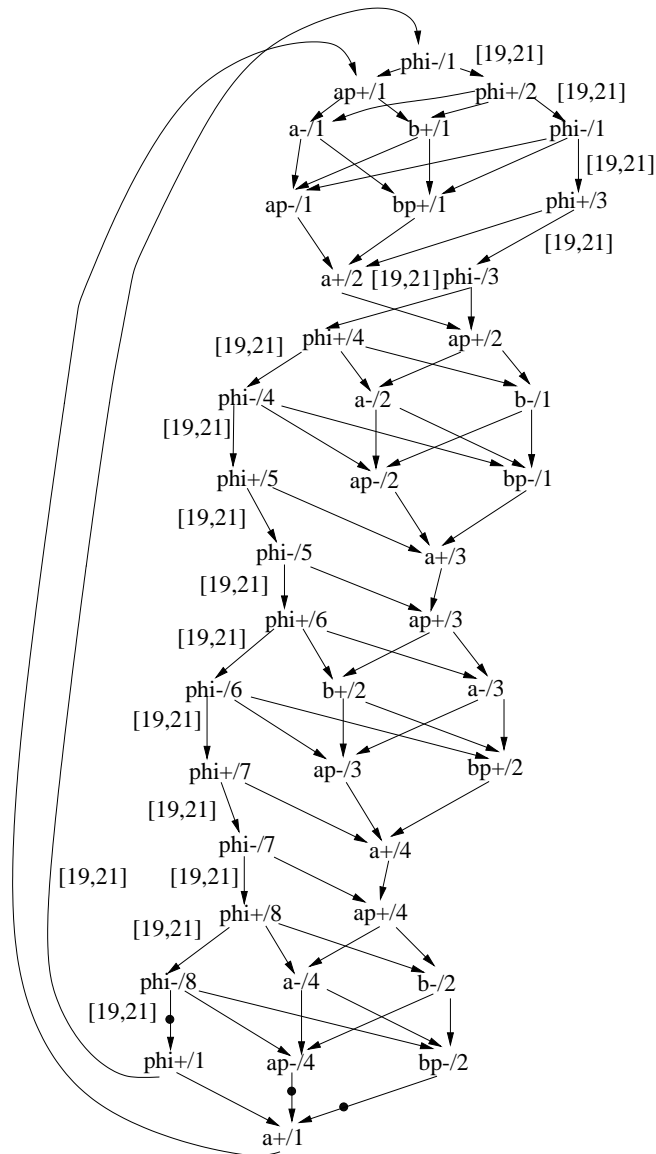


Figure 9.1. TEL structure for a 2-bit counter.

The first example is a n -bit synchronous counter. The basic operation of the counter is that when the clock goes high, the next value of the count is determined from the previous value. When the clock goes low, the new value is latched and fed back to determine the next count. The TEL structure for a 2-bit counter specification is shown

in Figure 9.1. If not otherwise indicated in the figure, the rules have a time bound of $[0, 5]$. Figure 9.1 shows that this example has several events which are enabled by multiple behavioral rules. In [50], Myers describes graph transformations that create a new specification which satisfies the single behavioral rule restriction allowing verification by *Orbits* [58, 59]. The counter could be specified more compactly if boolean expressions are used in the TEL structure, but *Orbits* cannot analyze a specification with boolean expressions. *ATACS* must be run on purely event based specifications in order to make comparisons to *Orbits*. Results when boolean expressions are used are shown in later sections.

Table 9.1 shows runtimes and regions generated using *ATACS* and *Orbits* for counters ranging in size from 2 bits to 7 bits. The results using different combinations of optimizations in *ATACS* are indicated in the tables as follows: “Geom” indicates the geometric algorithm presented in Section 3 without any optimizations. “Geom+All” indicates the geometric algorithm with all optimizations on. “PO” indicates the POSET algorithm without any optimizations. “Sub/sup” indicates the POSET algorithm with the subset and superset optimizations. “Inter” indicates that only the interleaving optimization is used, and “all” indicates that subsets, supersets, and interleaving are used. The last column, “Orbits”, gives the results of running *Orbits*. *Orbits* also contains many optimizations, all of which are used for this comparison. Entries of “mem” in the table indicate that the machine, a 400MHz Pentium II with 384MB of physical memory and 768Mb of swap space, runs out of memory. The example size is indicated next to the example name, where “E” represents the number of events and “R” represents the number of rules. Runtime comparisons are difficult between *ATACS* and *Orbits* since *ATACS* is implemented in C and *Orbits* is implemented in Scheme. Although *Orbits* is run on a compiled version of Scheme, which is much faster than interpreted Scheme, its runtimes are still degraded by the differences in implementation language. For this reason, differences in regions generated are useful to compare the algorithms in an implementation independent way.

The maximum counter size that *Orbits* can analyze is 3 bits. *Orbits* requires 1648 seconds and 10,222 regions to analyze a 3 bit counter, while the POSET algorithm with all optimizations can analyze a 3 bit counter in .07 seconds and 89 regions. This dramatic difference in region count and runtime occurs because the graph transformation adds $n!$ new events for each event that has n behavioral rules. In the 3-bit counter most of the

Runtimes for counters (in seconds)							
Example E/R	geom	geom+All	PO	sub/sup	inter	all	Orbits
cnt2 40/77	.08	.04	.07	.07	.07	.07	5
cnt3 45/98	17	.08	2	2	.07	.07	1648
cnt4 93/215	mem	1.1	mem	mem	.73	.73	mem
cnt5 189/453	mem	19	mem	mem	19	19	mem
cnt6 381/929	mem	250	mem	mem	280	280	mem
cnt7 765/1886	mem	2436	mem	mem	1945	1945	mem
Regions generated for counters							
Example E/R	geom	geom+All	PO	sub/sup	inter	all	Orbits
cnt2 40/77	211	57	171	168	57	49	240
cnt3 45/98	5687	89	1627	1620	89	89	10222
cnt4 93/215	mem	257	mem	mem	257	257	mem
cnt5 189/453	mem	705	mem	mem	705	705	mem
cnt6 381/929	mem	1857	mem	mem	1857	1857	mem
cnt7 765/1886	mem	4737	mem	mem	4737	4737	mem

Table 9.1. Results for counters.

events are enabled by 4 rules, causing a huge combinatorial explosion in the number of regions produced by `Orbits`. This example also shows the impact of the interleaving optimization. For a 3 bit counter, the interleaving optimization reduces the region count from 1627 regions to 89 regions, and allows the algorithm to analyze up to a 7 bit counter without running out of memory. Since the number of events enabled by many rules is high in this example, eliminating unnecessary rule firing interleavings produces a dramatic reduction in regions and runtime. This example also has another interesting result. Table 9.1 shows that the optimizations have a much greater effect than the POSET algorithm on improving performance. Although the POSET algorithm does perform better than the geometric algorithm when they are both run without optimizations, the region counts are identical when the geometric algorithm and the POSET algorithm are run with optimizations. This occurs because the interleaving optimization is able to eliminate rule firing sequences that result in region splitting by the geometric algorithm.

The next example is an asynchronous fifo composed of lazy-active/passive buffers (LAPB). These buffers perform one communication on their read port to receive a new data value, followed by another communication on their write port to send the value on to the next stage. The TEL structure for one stage of the LAPB is shown in Figure 9.2. Rules which do not have a timing bound in the figure have a bound of [1,5]. The values of

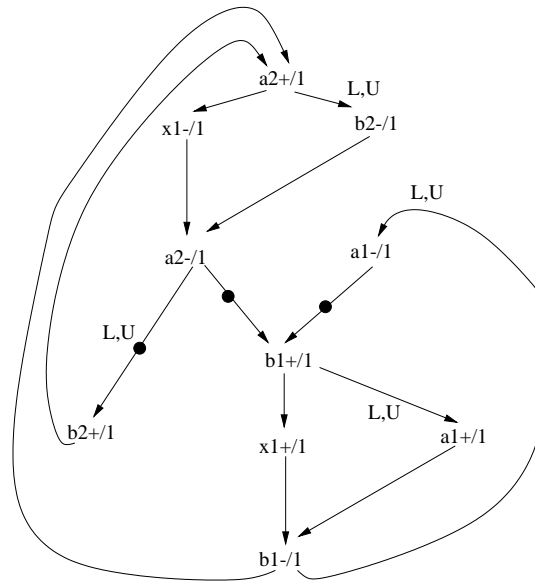


Figure 9.2. TEL structure for one LAPB stage.

Runtimes for LAPBs (in seconds)							
Example E/R	geom	geom+All	PO	sub/sup	inter	all	Orbits
LAPB1 11/21	.02	.01	.01	.01	.006	.006	.13
LAPB2 17/32	101	.2	.8	.5	.2	.2	2.5
LAPB3 23/44	mem	1.4	13	7	1.4	1.2	34
LAPB4 29/54	mem	50	mem	140	19	13	485
LAPB5 35/65	mem	6429	mem	mem	mem	100	mem
LAPB6 41/76	mem	mem	mem	mem	mem	654	mem
Regions generated for LAPBs							
Example E/R	geom	geom+All	PO	sub/sup	inter	all	Orbits
LAPB1 11/21	120	29	58	42	36	29	42
LAPB2 17/32	39,536	463	873	538	293	237	464
LAPB3 23/43	mem	2689	10,691	4500	1270	949	3271
LAPB4 29/54	mem	22,418	mem	40,970	7494	4574	22,504
LAPB5 35/65	mem	298,502	mem	mem	mem	25,419	mem
LAPB6 41/76	mem	mem	mem	mem	mem	140,663	mem

Table 9.2. Results for the LAPBs.

L and U used in the figure vary depending on where in the LAPB the stage occurs. If it is communicating with another *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. When many LAPB stages are composed together the resulting specification has many events that are enabled by multiple behavioral rules. The results generated for LAPB's ranging in length from 1 stage to 6 stages are shown in Figure 9.2. The longest LAPB that `Orbits` can analyze consists of 4 buffers and requires 22,504 geometric regions and 485 seconds. The analysis of a LAPB with 4 buffers using the POSET algorithm and all optimizations requires 4188 geometric regions and 17 seconds. The POSET algorithm can analyze up to six buffers. Also, in this example, the impact of the POSET algorithm is much greater than the impact of the optimizations. For the largest example where both algorithms complete, the geometric algorithm with optimization is an order of magnitude slower and generates an order of magnitude more regions than the POSET algorithm with optimizations.

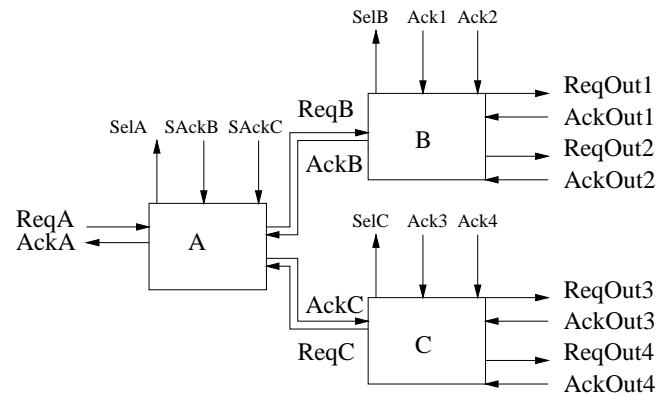


Figure 9.3. 2 level selector.

The next example is the two level selector circuit shown in Figure 9.3. The circuit first receives a request on the *ReqA* wire. This causes module *A* to send a request on the *SelA* wire. It receives a response either on the *SAckB* wire or the *SAckC* wire. Module *A* then sends a request on either the *ReqB* or the *ReqC* wires, depending on which response is received for the *SelA* request. For example, when module *B* receives the request on *ReqB*, it sends a request on *SelB*. The response determines whether module *B* initiates a communication on *ReqOut1* or *ReqOut2*. When its output communication is complete, it sends an acknowledge on *AckB*. This allows module *A* to acknowledge that the selection

Runtimes for Selectors (in seconds)									
Ex. E/R	geom	g+A	PO	sub/sup	inter	all	-M	app.	Orbits
sel1 18/31	.3	.02	.25	.1	.03	.03	.04	.03	.6
sel2 37/76	mem	44	mem	44	11	5	7	5	152
sel3 23/44	mem	mem	mem	mem	mem	587	710	295	mem
Regions generated for Selectors									
Ex. E/R	geom	g+A	PO	sub/sup	inter	all	-M	app.	Orbits
sel1 18/31	793	64	378	187	63	58	68	58	133
sel2 37/76	mem	13213	mem	9476	4479	1736	2139	1711	5417
sel3 53/110	mem	mem	mem	mem	mem	50320	67582	40291	mem

Table 9.3. Results for the selector unit

The results for this example are shown in Table 9.3. Since this example has conflict, two additional columns are added. The first is “-M”. This column gives results if the merging optimization is *not* used. All of the other columns contain results based on using the merging optimization along with the specified one. The results in the “-M” column are computed with all of the other optimizations on. The second additional column is “approx”. It is added to the table to show the results when the conflict restriction in the POSET algorithm is removed. When the “approx” option is used, the algorithm does not check to see if an event is enabled by a rule with a non-empty choice set when computing upper bounds in the POSET matrix. All of the other optimizations are also used with this approximation. In this example and the next example, the set of reachable untimed states found with this approximation is the same as the set of untimed states found with the exact algorithm. There is an improvement in runtime on the order of 40% when the approximation is used on the largest example. This shows that the choice restriction is adding extra regions and degrading performance somewhat, but that the effect is not dramatic. If conservative results are acceptable, this approximation can be used to improve performance. If conservative results are not acceptable the runtime penalty to achieve exact results is not prohibitive.

Table 9.3 also shows that the POSET algorithm in *ATACS* compares favorably with *Orbits*. *Orbits* requires 152 seconds and 5417 regions to analyze the two selectors version, while the exact POSET algorithm with all optimizations requires only 1736 regions. For the full circuit with both *B* and *C* blocks included, the POSET algorithm completes the analysis, using 54,725 regions, and *Orbits* runs out of memory and does

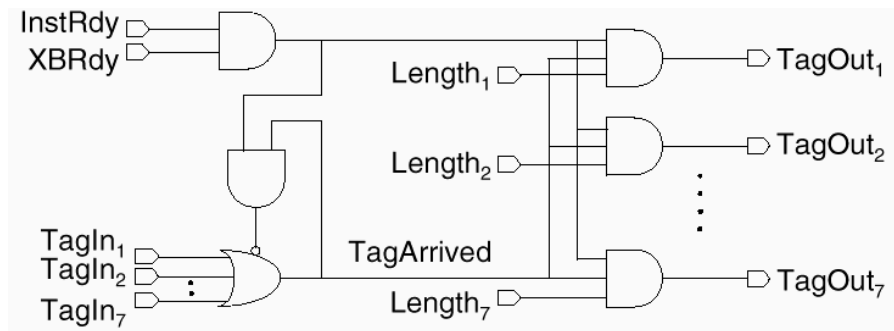


Figure 9.5. The tag unit circuit.

not complete. These results show that even when the algorithm restricts regions when conflicts occur, it still generates many fewer regions than **Orbits**. The results also show that the merge optimization contributes to a significant increase in performance.

The final example comes from the Intel RAPPID design [60]. The RAPPID design is a fully asynchronous instruction length decoder for the x86 instruction set. This design is shown to be 3 times faster while using half the power of a corresponding synchronous design from a 400 MHz x86 processor. The key to the performance is a very efficient synchronization mechanism which is called the *tagunit*. One tagunit is shown in Figure 9.5. The operation of this circuit is that it can receive a tag from one of seven other tag units ($TagIn_i$). If the instruction is ready ($InstRdy$) and the crossbar is ready ($XBRdy$), it tags out to one of seven other tag units ($TagOut_i$) depending on the length of the instruction ($Length_i$). The the tagunit is checked for hazard-freedom using **ATACS** and **Orbits**, and the results are shown in Table 9.4. In order to parameterize the example, we verified *tagunits* of various sizes where the size is the number of units from which a tag could be received and to which a tag can be transmitted. The *tagunit* specification contains many rules with non-empty choice sets, and the impact of the choice restriction is illustrated using the approximation described previously. The result of the approximation in the *tagunit* is similar to the result in the selector. Removing the choice restriction produces approximately a 40% improvement in runtime for the largest *tagunit*. Unlike the selector, **Orbits** completes the largest *tagunit* specification. **Orbits** does not fail due to state explosion in this example, but **ATACS** with all optimizations produces approximately one third the regions that **Orbits** produces for all sizes of *tagunit*

except size one. This example has fewer events which are enabled by large numbers of rules, which explains the improved performance of `Orbits`. In this example the merge optimization is key to good performance. The specification for the *tagunit* contains very large merges, and for the larger tag unit, the merge optimization is responsible for an order of magnitude performance improvement in both speed and runtimes. Without the merge optimization, `ATACS` is not competitive to `Orbits`, which operates on timed Petri nets.

Runtimes for tag units (in seconds)									
Ex. E/R	geom	g+A	PO	sub/sup	inter	all	-M	app.	Orbits
tag1 17/42	53	.4	1	.6	.3	.2	.2	.2	3.2
tag2 25/69	mem	2.5	18	21	2.2	1.7	4.3	1.3	35
tag3 33/98	mem	9.4	72	37	7	5.5	30	4.1	66
tag4 41/134	mem	25	144	95	14	12	90	9	107
tag5 49/188	mem	65	mem	199	37	34	302	22	162
tag6 57/242	mem	135	mem	mem	57	24	697	37	229
tag7 65/304	mem	286	mem	mem	103	103	1871	69	284
Regions generated for tag units									
Ex. E/R	geom	g+A	PO	sub/sup	inter	all	-M	app.	Orbits
tag1 17/42	20077	717	1133	619	378	253	253	253	442
tag2 25/69	mem	1766	8127	3696	1011	676	1965	622	2751
tag3 33/98	mem	3018	14265	6603	1755	1186	5903	1085	4816
tag4 41/134	mem	4688	21956	10391	2680	1831	12912	1671	7409
tag5 49/188	mem	6461	mem	14844	3771	2596	24523	2365	10530
tag6 57/242	mem	8773	mem	mem	5044	3497	40836	3183	14179
tag7 65/304	mem	11051	mem	mem	6483	4518	64191	4109	18356

Table 9.4. Results for tagunit.

The *tagunit* also provides a good example of the improvement that is gained by representing a circuit in the more concise level based form. Table 9.5 shows that the level based tag unit requires less time and fewer regions than the event based tag unit for all algorithm and optimization combinations. It also shows that the impact of all optimizations are not as dramatic on the level based specification, and that the interleaving optimization produces no benefit since it cannot be applied when rules have level expressions. When all optimizations are used, the POSET algorithm completes analysis on the seven stage, level based tag unit in 13 seconds, using 1246 regions. This is

nearly a five times improvement in runtime and region count over the POSET algorithm with all optimizations on the event based specification.

Runtimes for tag units (in seconds)									
Ex. E/R	geom	g+A	PO	sub/sup	inter	all	-M	app.	Orbits
tag7 65/304	mem	286	mem	mem	103	103	1871	69	284
l_tag7 101/180	117	13	28	17	28	17	79	13	n/a
Regions generated for tag units									
Ex. E/R	geom	g+A	PO	sub/sup	inter	all	-M	app.	Orbits
tag7 65/304	mem	11051	mem	mem	6483	4518	64191	4109	18356
l_tag7 101/180	9151	2089	1727	1246	1727	1246	2389	1246	n/a

Table 9.5. Comparison with level based tag unit

In our experience, **ATACS** with all of the optimizations performs better than **Orbits** in all specifications that have multiple behavioral rules. **ATACS** is also an improvement over **Orbits** since it can analyze level based specification which more concisely represent the circuit. If a specification does not have multiple behavioral rules or level expressions, the **ATACS** algorithm and the **Orbits** algorithm produce similar results.

9.2 Comparison with Other Verification Methods

Geometric region based timing analysis is often dismissed as impractical due to its performance on highly concurrent examples. These algorithms do perform quite poorly compared to other algorithms if the POSET approach is not used. However, this section shows that the POSET algorithm far outperforms other approaches on highly concurrent specifications.

The first two examples, Alpha and Beta shown in Figure 9.6, are presented by Bozga in [13]. Each stage of the Alpha example is composed of a single event which can fire repeatedly at a given interval and is not effected by any other events in the system. The authors of [13] show that techniques based on DBMs (i.e., geometric regions) can only handle 5 stages of this highly concurrent example while their symbolic discrete-time technique using numerical decision diagrams (NDDs) can handle 18 stages in 12 hours on a SUN UltraSparc with 256MB of memory. A *loglog* plot of the results from [13] and our results using POSET timing on a SPARC 20 with 128 MB of memory are shown in Figure 9.7. These results indicate that POSET timing is orders of magnitude faster

and more memory efficient. In fact, our techniques found the reachable states space for 512 stages in about 73 minutes using 112 MB of memory. This simple example clearly has only one untimed state regardless of the number of stages, and POSET timing can represent the timed state space using only one geometric region. Our technique does not find the region in its first iteration, however. It first finds a number of smaller regions before finding the final region that is a superset of all the rest. Therefore, although its performance is very good, it does not analyze the example instantaneously.

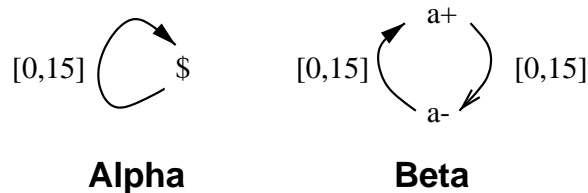


Figure 9.6. TEL structures for the Alpha and Beta examples.

One stage of the Beta example is composed of one state bit per stage with two events, one to set and one to reset the bit. In [13], Bozga shows that DBMs can only handle 4 stages while their technique can handle 9 stages. A semilog plot of their results and ours are shown in Figure 9.8. POSET timing can handle 14 stages in 108 MB of memory in just 16 minutes. For the Beta example, the number of states is exactly 2^n where n is the number of stages, so POSET timing could handle an example with 32 times more untimed states than in [13]. Again, POSET timing is able to represent all the timing behavior in this example using one geometric region per state. Clearly, the Alpha and Beta examples are ideally suited to our algorithm, but they are used in [13] to demonstrate the weakness of traditional geometric region based methods.

The last example is a STARI communication circuit described in detail by Greenstreet in [34, 33]. The STARI circuit is used to communicate between two synchronous systems that are operating at the same clock frequency, π , but are out-of-phase due to clock skew which can vary from 0 to *skew*. The TEL structure for the environment of this circuit is composed of a clk process (Figure 9.9(a)), a transmitter (Figure 9.9(b)), and a receiver (Figure 9.9(c)). The STARI circuit is composed of a number of FIFO stages built from 2 C-elements and 1 NOR-gate per stage (Figure 9.9(d)), each of which has a delay bound

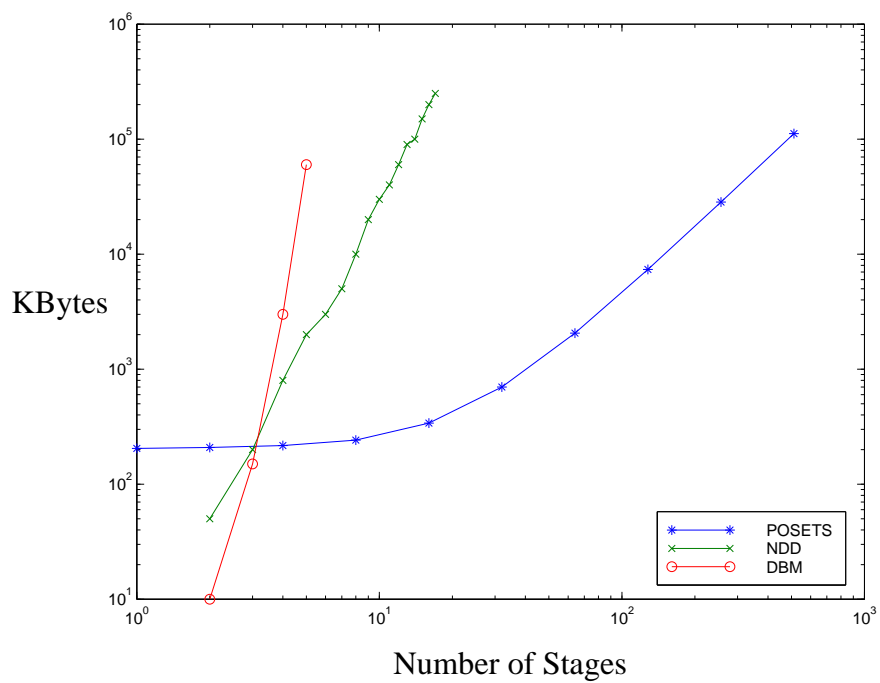
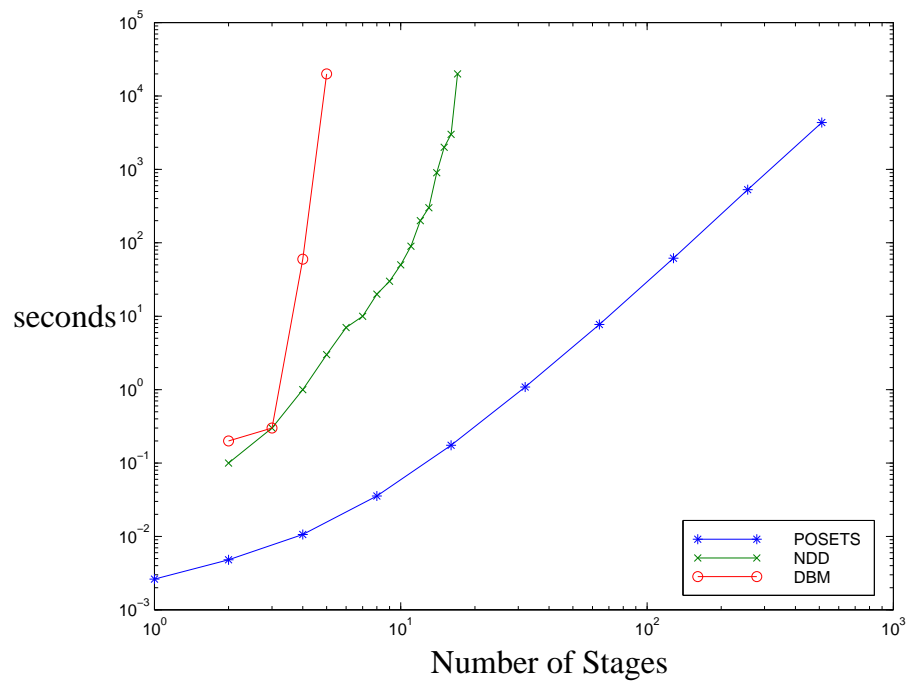


Figure 9.7. Comparative performance for the Alpha example.

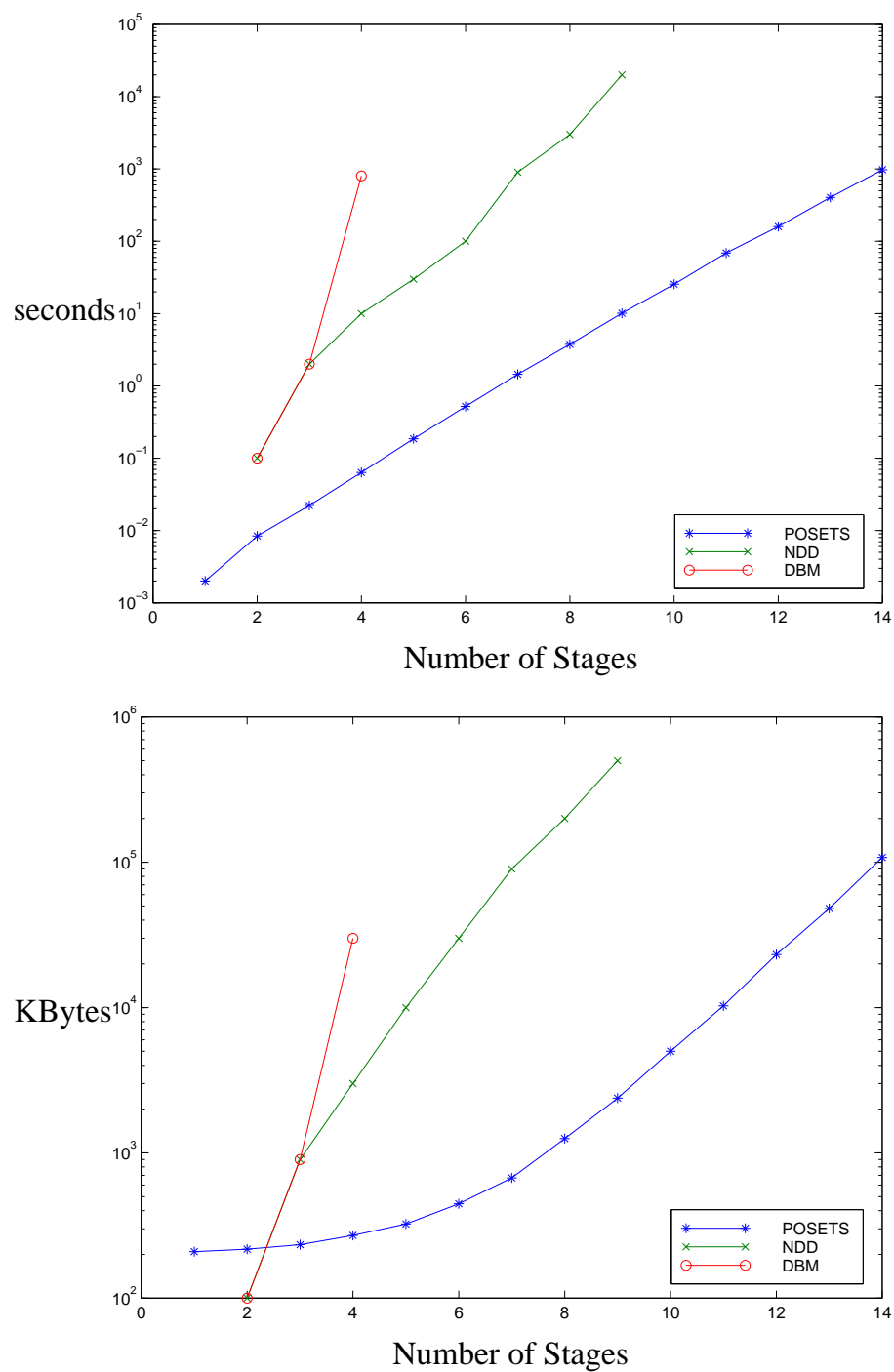


Figure 9.8. Comparative performance for the Beta example.

of $[l, u]$. There are two properties that need to be verified: (1) each data value output by the transmitter must be inserted into the FIFO before the next one is output (i.e., $ack(1)-$ precedes $x(0).t-$ and $x(0).f-$) and (2) a new data value must be output by the FIFO before each acknowledgment from the receiver (i.e., $x(n).t+$ or $x(n).f+$ precedes $ack(n+1)-$) [64]. To guarantee the second property, it is necessary to initialize the FIFO to be approximately half-full [33]. In addition to these two properties, we also verified that every gate is hazard-free (i.e., once a gate is enabled, it cannot be disabled until it has fired).

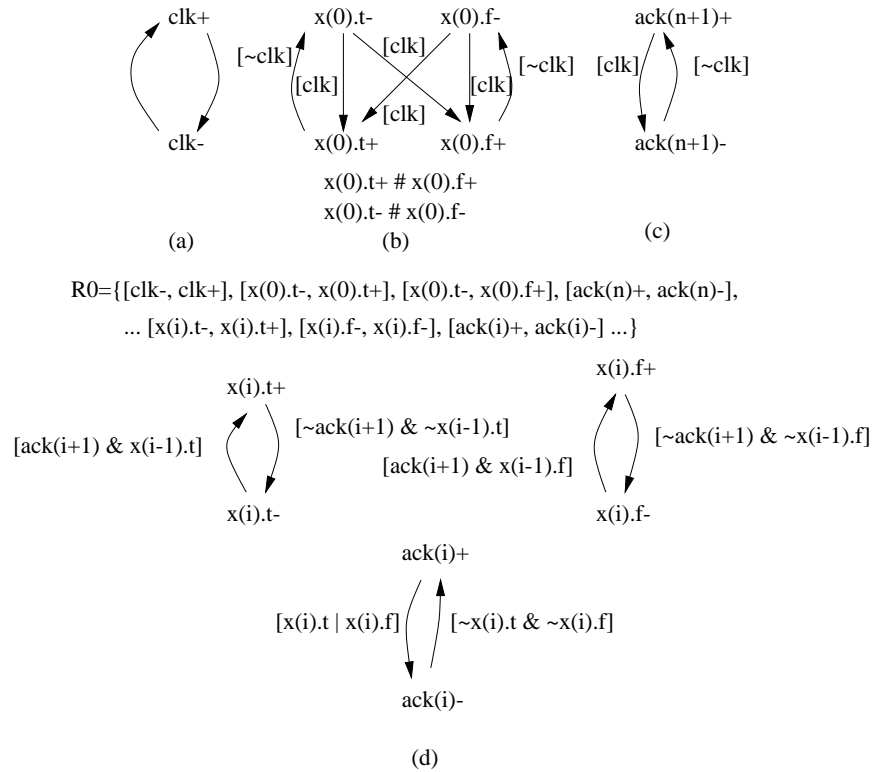


Figure 9.9. TEL structures for the STARI example (a) the clock process with timing constraints $= [\pi, \pi]$; (b) the transmitter process and (c) the receiver process with timing constraints $= [0, skew]$; and (d) a STARI FIFO stage with timing constraints $= [l, u]$.

There have been nice proofs of STARI's correctness by Greenstreet [33] and Hulgard [38], but they have been on abstract models. In [64], Tasiran states that COSPAN, which uses the unit-cube (or region) technique for timing verification [6], runs out of

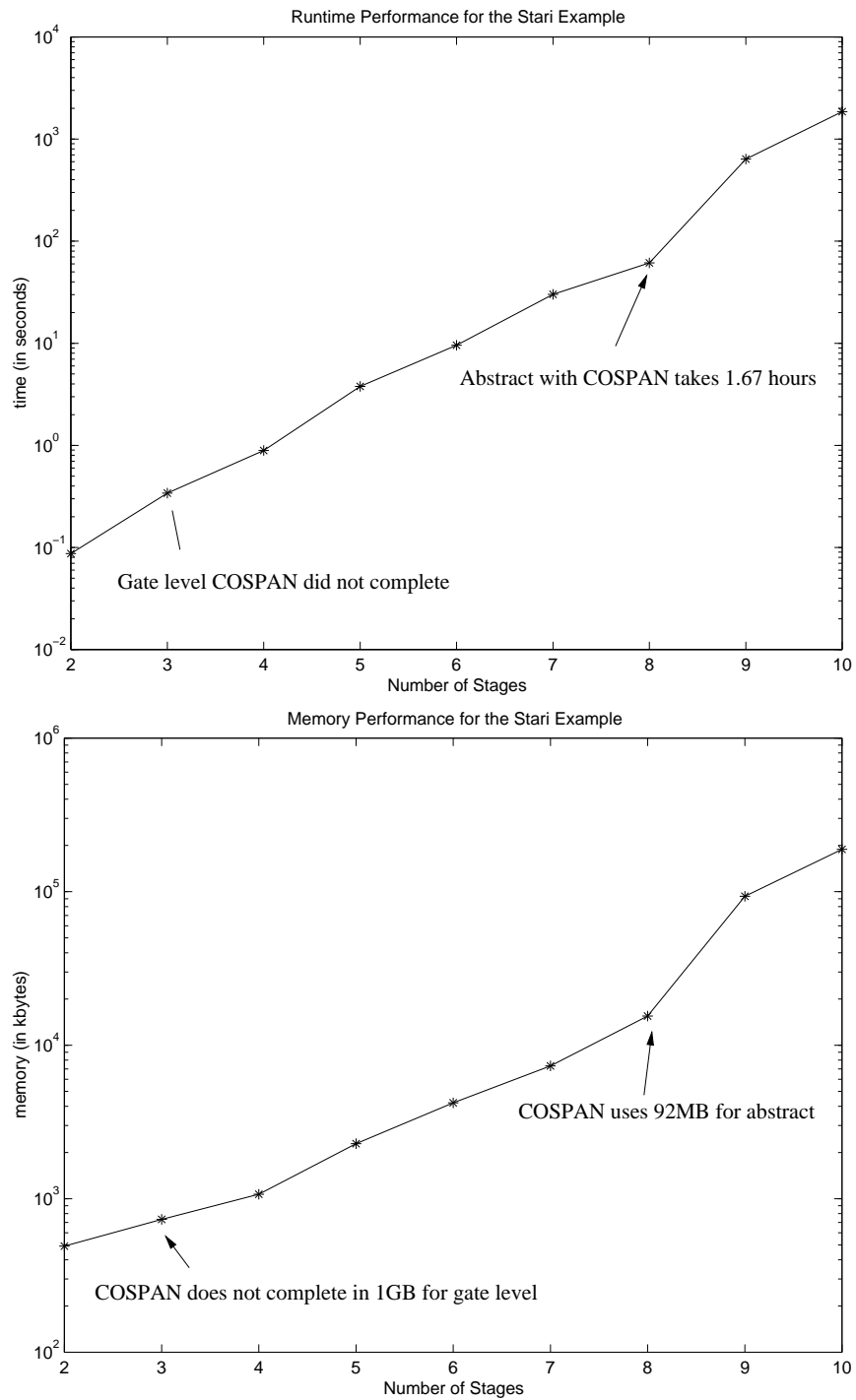


Figure 9.10. Stari results with POSETs and with COSPAN.

memory attempting to verify a 3 stage gate-level version of STARI on a machine with 1 GB of memory. This paper goes on to describe an abstract model of STARI for which they could verify 8 stages in 92.4 MB of memory and 1.67 hours. We first verified STARI at the gate-level with delays from [64] (i.e., $\pi = 12$, $skew = 1$, $l = 1$, and $u = 2$). Using POSET timing, we can verify a 3 stage STARI in 0.74 MB in only 0.40 seconds. For an 8 stage STARI, the verification took 12 MB and only 55 seconds. In fact, POSET timing could verify 10 stages in 124 MB of memory in less than 20 minutes. This shows a nice improvement over the abstraction method and a dramatic improvement over the gate-level verification in `COSPAN`. For 10 stages, POSET timing found 14,529 untimed states and only needed 15,349 geometric regions to describe the timed state space. This represents a ratio of only 1.06 geometric regions per untimed state.

Finally, the complexity of POSET timing is relatively independent of the timing bounds used. We also ran our experiments using $l = 97$ and $u = 201$, $skew = 101$, and $\pi = 1193$ which found more untimed states. With $l = 102$, we found less untimed states. Both cases with higher precision delay numbers had comparable performance to the one with lower precision delay numbers. This shows that higher precision timing bounds can be efficiently verified and can lead to different behaviors. It would not be possible to use this level of precision with a discrete-time or unit-cube based technique, since the number of states would explode with such large numbers.

9.3 Implicit Methods

This section describes the memory improvements that are achieved by the application of the MTBDD region representation described in Chapter 7. Results are presented on a parameterized version of the high-performance FIFO element described by Molnar in [48] and the arbiter presented by Greenstreet in [32]. These specifications are highly concurrent and produce a large number of geometric regions.

Figure 9.11 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.

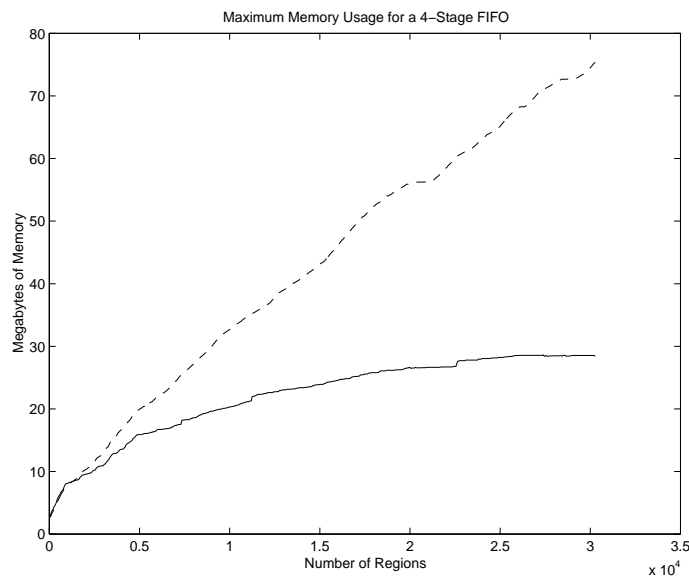


Figure 9.11. FIFO memory performance.

The second example is the arbiter design presented in [32]. Most arbiters are designed to minimize the probability of a metastability failure. However, since metastability is rare, this arbiter is designed to maximize performance when there is no metastability. It does so by using a highly concurrent, timed design. Instead of a standard four-phase handshake, this arbiter uses an *asP** protocol [48] which uses timed pulses for communication. The arbiter only works if the timing on the pulses is correct. The timed state space of the arbiter is very large due to its high concurrency. The TEL structure representing the arbiter has 30 signals, 67 events and 73 rules. Since the number of rules is high, the matrices representing these regions are quite large and use a lot of memory. The large

state space and the large size of the regions make the arbiter an excellent candidate for the use of MTBDDs to improve memory performance.

The number of reachable boolean states is highly dependent on the delay ranges specified. Two versions of the arbiter are used to measure the MTBDD improvement. The delays in the first version result in the generation of 22,953 untimed states and 101,273 regions when the POSET algorithm is used. State space exploration requires ~ 550 Mb of memory using the explicit representation and ~ 130 Mb of memory using the MTBDD representation, which is slightly more than a 3 times memory improvement. Unfortunately, the MTBDD representation results in an order of magnitude degradation in runtime. It takes ~ 9000 seconds to complete compared to ~ 2800 seconds for the explicit representation. This is not a good tradeoff if there is sufficient memory to complete the state space exploration using the explicit method. However, the experiment using the arbiter with the larger state space shows that the MTBDD method can allow larger examples to be analyzed. By increasing the delay ranges, the state space size of the arbiter is increased to 45,552 untimed states and 149,708 regions. On this example, the explicit method runs out of memory after consuming over 850Mb while the MTBDD method complete using ~ 300 Mb and $\sim 17,000$ seconds. These results show that, while slow, the MTBDD method does allow examples to complete that could not complete using the explicit method.

9.4 Application to Synchronous Circuits

The POSET algorithm in ATACS has been used to analyze several circuits from the guTS (gigahertz unit Test Site) integer microprocessor designed at IBM's Austin Research Laboratory and presented by Hofstee in [35]. The results of this experiment, which we first first present in [10], demonstrate that asynchronous analysis techniques can be used to fill gaps in synchronous design methodology for highly timed synchronous circuits.

The purpose of the guTS design is to demonstrate the performance gains that can be achieved using aggressive circuit design. It is implemented in a 0.25μ CMOS process available in 1997. The high-performance of the circuit is a result of the circuit design, which is done in a dynamic circuit style known as delayed-reset domino and described by Nokwa in [55] and Chappell in [21]. Although TEL structures and the POSET algorithm were originally developed to analyze asynchronous circuits, they are well suited to the analysis of delayed-reset domino circuits.

The guTS microprocessor contains a set of macros which operate synchronously. A delayed-reset domino macro consists of a number of levels of dynamic gates, each of which receives inputs from preceding layers. Standard domino gates use a common clock that acts as a timing reference. In a delayed-reset design, each level of dynamic gates receives its own, precisely timed clock, which is generated by a buffer chain within the macro. The local clocks travel through the logic along with the data, a reset wave preceding each computation wave. This technique allows approximately one-half cycle for each gate to reset and one-half cycle for each gate to evaluate. The cycle time for a delayed-reset domino macro is set by adding the necessary precharge and evaluate times for a single gate. If multiple gates operate on the same precharge signal, cycle time is set by adding the evaluate delay through all the stages to the precharge delay. Due to the overlapping of the precharge and evaluate phases, the delayed-reset domino approach significantly increases the amount of dynamic-logic that can be placed in a macro at a given clock frequency.

The delayed-reset domino gates used in the guTS processor lack the “foot” device that is included in a standard domino gate. The purpose of this device is to turn off the gates’ pulldown stack during the precharge phase. Removing this device allows the gate to switch 5% to 15% faster. Alternatively, the gate can compute a more complex logic function using the same transistor stack height [55]. In order to remove this transistor, it is necessary to ensure that the evaluate logic is not on during the precharge phase. This is the case if all inputs to the gate are guaranteed to be low during the precharge phase. To meet this requirement, the inputs to the macro must be pulsed. Combined with the requirement that the inputs to each gate remain stable high long enough to switch the dynamic node, this results in a two sided timing verification problem which is unusual for a synchronous design.

In the guTS processor, the macro level timing verification is done using extensive SPICE level circuit simulation [57]. After the delay behavior of the macros is characterized by designers in SPICE, it is incorporated into a chip level timing model for chip level static timing verification. This was a successful approach for this processor since it worked in first silicon. However, in order to ensure the correctness of the processor over all variations in delay, large amounts of delay margin are included in the design of the macros. If it is possible to formally verify the macros, less margin is necessary to have confidence in the processor’s correctness, which IBM designers estimate may result in

performance improvements up to 10%. The timing constraints that need to be checked in the delayed reset domino macros are very similar to the correctness constraints necessary for asynchronous circuits, and the delayed reset domino circuits are quite similar to asynchronous circuits. Therefore, an asynchronous timing verification tool is a natural choice to be used for formal verification of the macros.

9.4.1 Verification of Gate Level Models

ATACS is used to verify several of the macros from the guTS processor. The first circuit is a combined multiplexor and latch (MLE). This circuit is small enough to verify at the gate level, and is shown in Figure 9.12. The goal with this circuit is to verify that the timing specification which is supplied with the circuit indeed guarantees that the circuit works correctly. The timing specification describes the timing requirements that must be met by any circuit communicating with the MLE. It is derived from SPICE level simulation and the circuit designers knowledge of how the circuit works. The timing specifications are also used as the basis for chip level static timing analysis. In order to ensure that the chip-level static timing analysis is modeling all timing behavior, each macro needs to be formally verified in the environment described by the timing specification. ATACS verifies the MLE circuit in a few seconds on a 400MHz Pentium II.

The MLE circuit contains both static and dynamic gates. The inputs to static gates are allowed to be unstable since this does not immediately cause a failure. However, if a glitch on the output of a static gate propagates to the input of a dynamic gate, it can cause a failure. In the MLE circuit, the gate driving the signal “output complement” is static. In every cycle where “output complement” does not fall, there is a glitch on its inputs. At the end of the precharge phase, the signal “Output_” is always high and it feeds one of the inputs to the static gate. When the clock rises, “output complement” always begins to fall. However, the signal “Output_” falls later in the clock cycle if the selected data value is high. When “Output_” falls, one of the inputs to the static gate is driven low and “output complement” rises again, producing a glitch. ATACS detects this glitch and determines that it cannot propagate to the output of the circuit.

The next circuit is a dynamic PLA that is used in the processor’s control circuitry. Dynamic PLAs are easy to generate automatically and have predictable area and delay. In order to make the PLAs fast, they are controlled using self-resetting circuitry. An example of the control circuitry is shown in Figure 9.13. The circuit uses a very aggressive

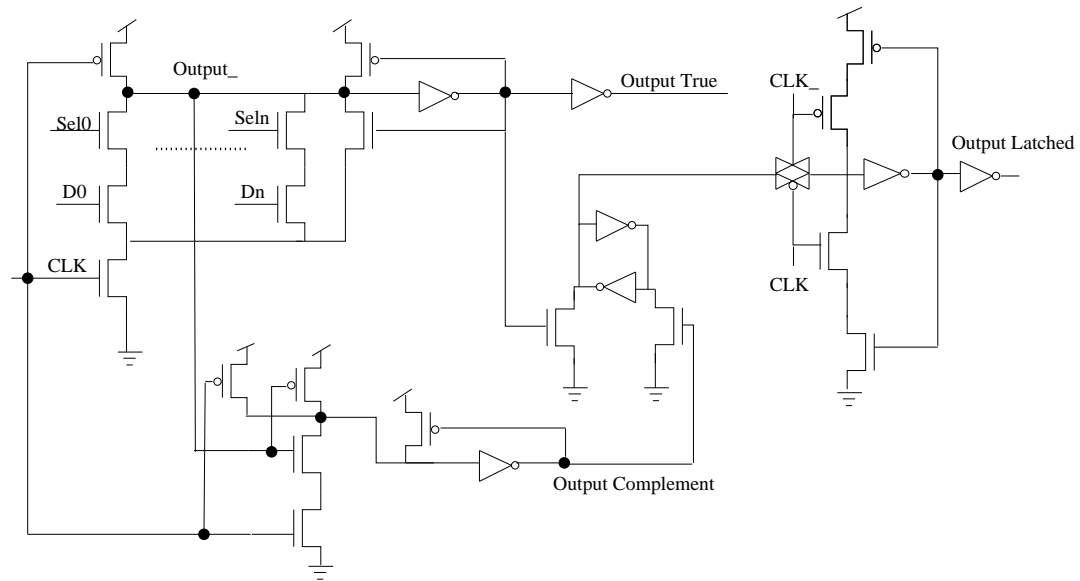


Figure 9.12. The MLE circuit.

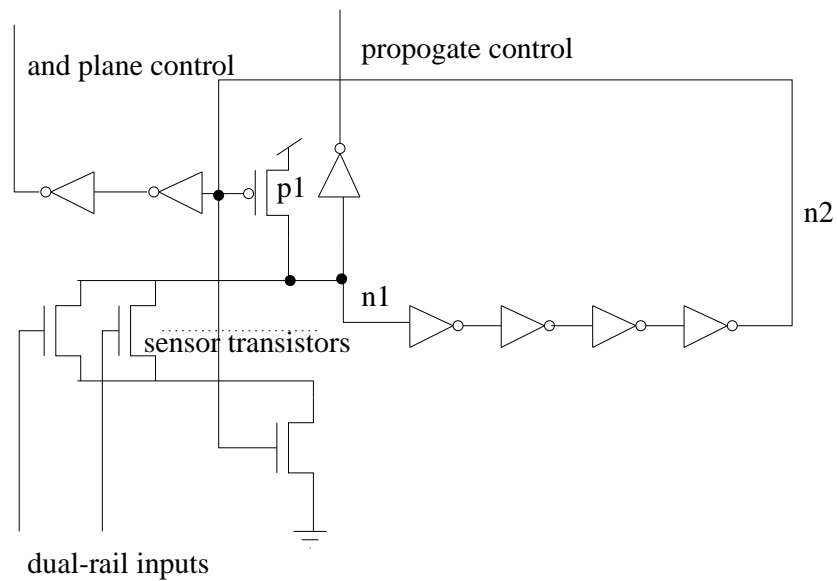
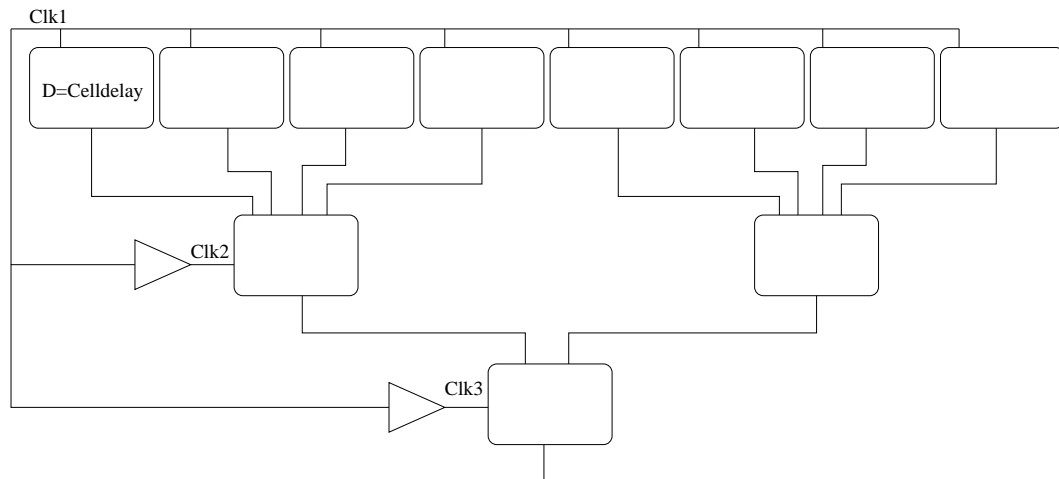


Figure 9.13. PLA control.

technique to determine when its inputs are valid. The inputs are presented to the circuit dual-rail. When the inputs are valid the sensor transistors are turned on. These



Designed Celldelay = Evaluate:129 - 139, Precharge:149-153

Figure 9.14. Model for the compare unit.

transistors are all connected to a single node, $n1$, which has been precharged high. The sensor transistors are sized so that one of them must be turned on for each input in order for $n1$ to discharge quickly. However, if one input arrives much earlier than the rest, eventually its single sensor transistor can discharge $n1$, erroneously causing the PLA to begin evaluating early. This completion detection circuit is highly timing dependent and only works if the inputs are guaranteed to arrive within a narrow time interval. After the falling edge of $n1$ propagates through four inverters, the node $n2$ falls. When this node falls, transistor $p1$ is turned on which raises node $n1$, resetting the completion detection circuit. The line “and plane control” is used to gate transistors which determine if the and-plane of the PLA is in precharge or evaluate mode. The line “propagate control” is used in a similar manner to control whether the output of the and-plane can propagate to the or-plane of the PLA, which is not shown. This control circuitry is essentially asynchronous. Asynchronous, self-resetting circuits are difficult for static tools to handle since they often assume that a transition on an input causes only a single transition on an output. ATACS is able to verify the circuit using the designed delays in a few seconds.

9.4.2 Verification of Abstracted Models

The next circuit is a compare unit for two 64 bit quantities. It consists of 3 stages of delayed-reset domino logic. The logic in each stage is exactly the same. A stage consists of a set of blocks that produce an output which indicates whether its two four bit inputs

are equal. To do a 64 bit compare, a tree structure is used where the first stage has 16 logic blocks, the second stage has 4 logic blocks, and the final stage has 1 logic block. Unlike the previous two examples, this circuit is too large for ATACS to verify it using a representation derived directly from its transistor level schematic. However, with a small amount of abstraction, it can be verified quickly. It is not necessary to model each of the 64 bits entering the compare unit. Each block in the first level of logic is modeled as a gate that waits for a single input and produces its output some variable amount of time later. Variability in input signal arrival times is accounted for by putting an independent delay range on the arrival time of the abstracted input signal for each of the blocks in the first level of logic. When this signal rises in the abstracted model, it is equivalent to all eight input bits to a block becoming stable in the actual circuit. Additionally, since the timing behavior of each block is the same, the number of input blocks can be reduced from 16 to 8 without effecting the timing behavior of the circuit. Figure 9.14 shows the structure of the model. Each block is represented as a TEL structure which raises its output signal 129 to 139 time units after the block receives all of its inputs, and lowers its output 149 to 153 time units after its local clock falls. A global clock which controls the transition times of the local clocks is also modeled but not shown. It takes 3 seconds to explore the state space of this model using the POSET state space algorithm on a 400MHz Pentium II. This circuit example also demonstrates the advantages of the level based specification. A purely event based representation of the comparator takes 7 seconds to complete with the POSET algorithm and generates three times as many regions. The iteration time provided by the POSET algorithm makes it reasonable to iteratively adjust the Celldelay values, global clock speed, and local clock timings to determine the working ranges of the circuit under a variety of assumptions. The circuit verifies for global clock cycles up to 100ps less than the clock cycle necessary for correct operation in the Gigahertz processor.

The next example is the verification of the 64-bit adder portion of the Multifunction Fixed Point Unit (MFXU). This unit computes the results of the add, subtract, and compare instructions for the processor. The core of the unit is the 64-bit parallel prefix adder design presented by Silberman in [62], which is based on the algorithm described by Kogge in [39]. The MFXU adder contains five stages of delayed reset domino logic. The first stage contains a true/complement mux, stages two through four compute the propagate and generate signals for the adder, and the fifth stage implements a large mux which merges many different signals. Each block contains a few domino gates, which can

vary in delay. Attempts to verify this circuit at the gate level quickly use more than half of a gigabyte of memory and do not complete. However, a conservative abstraction of the MFXU verifies in ATACS using the POSET algorithm in about 3 minutes. The verification does not complete using the geometric algorithm.

The structure of the MFXU abstraction is shown in Figure 9.15. There are two steps involved in creating the conservative abstraction of the MFXU. The first is to reduce the complexity of each block by lumping the delay ranges for all of the different gates into one delay range which represents the minimum and maximum time difference between the block receiving all of its inputs and generating all of its outputs. For example, suppose a block contains two domino gates. One of the gates takes 100ps to evaluate and the other takes 150ps to evaluate. It is conservative to make a model for the block where the minimum evaluate time for the block is 100ps and the maximum evaluate time for the block is 150ps. This abstraction does not capture the gate level behavior that one output of the block is available after 100ps and the other is available after 150ps, but if a circuit verifies using the abstraction, its actual behavior verifies also. An abstraction like this is made for the precharge phase and the evaluate phase of each block. Then the number of blocks is decreased. The goal is to reduce the number of blocks, without hiding any interesting block interactions. This is done by analyzing a 32-bit wide slice of the design. Since each block operates on four bits of input, this corresponds to a model that is eight blocks wide at its input. This model is large enough to include all of the types of interblock relationships of the larger design, and is small enough to verify quickly.

This is done by starting at the last stage and working toward the first. Every block in the last stage is included. Then, for every block in the last stage, at least two instances of each type of block that provides inputs to the last stage are included in the fourth stage. In this case, four instances of the *row3gen* block which feeds *sumout* block in the fifth stage are included. Only one instance of the *halfsum* block is included since there is only one *halfsum* block in the complete circuit. This process is then repeated for the fourth through first stages. The resulting model represents a conservative model of the possible timing relationships in the circuit, and is small enough to verify quickly.

The circuit, abstracted in this way, verifies at its intended clock speed. Therefore, any gate-level timing relationships that are missed by the abstraction are not necessary in order for the circuit to run at the specified speed. If this is not the case, then the blocks on the failure path can be specified in more detail. Although this increases verification

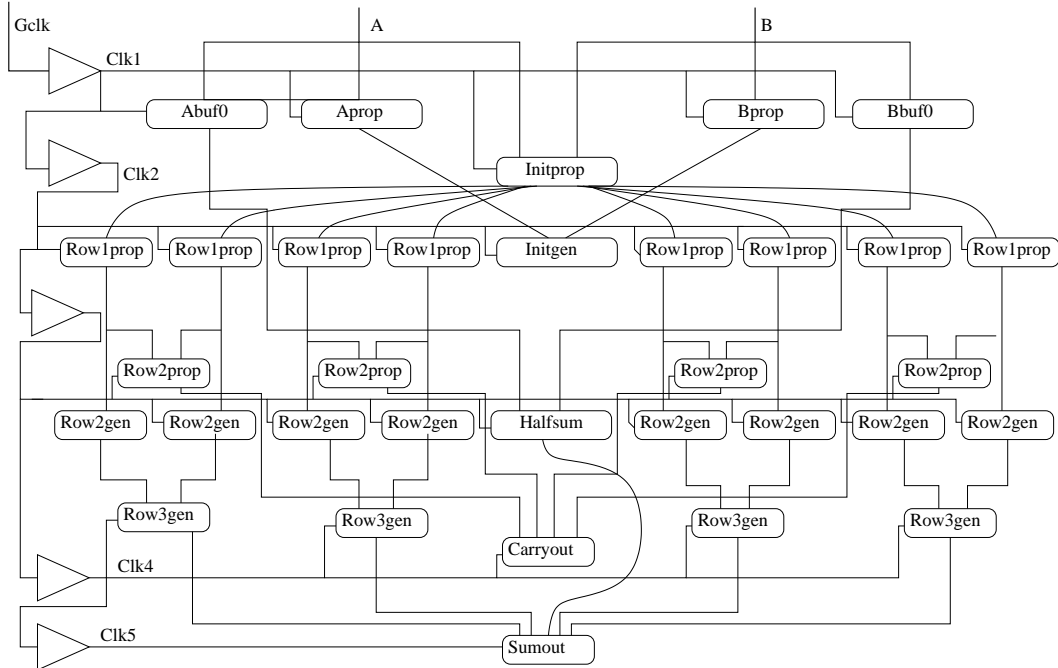


Figure 9.15. MFXU structure.

time, it should not make the problem intractable since the additional detail is limited to a few blocks. Even the abstracted version of this circuit is quite large, it has complex timing relationships which provide many possibilities for error. Formal verification gives confidence that all of the timing behaviors have been considered. Currently, ATACS does not have an automated method for generating circuit abstractions, and the abstraction described for this example is done manually. It may be possible to adapt techniques presented by Kikimoto in [40] to develop an automated method for abstracting blocks of domino gates.

The final circuit is an arithmetic circuit used in the integer execution unit. It is of moderate complexity and therefore can be used to test the accuracy of an abstracted model compared to a gate level model. The gate level model is still somewhat abstract in that it does not include the full 64-bit datapath, but each instance of a block is described at the gate level. The results on this macro indicate that the limiting factor in clock speed is the time that the inputs arrive to the macro, not gate to gate interactions inside the macro. Because of this, the maximum clock speeds allowed by the abstracted model

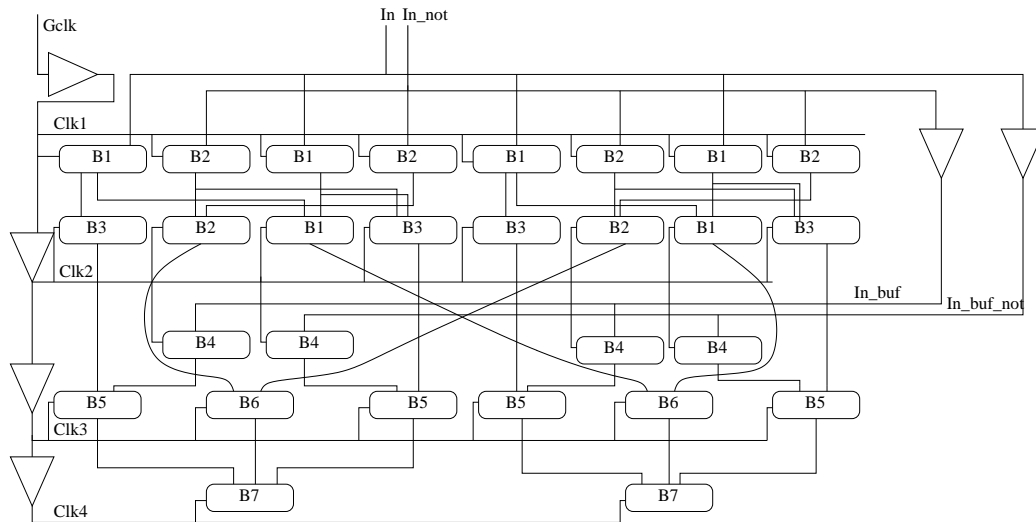


Figure 9.16. The CLZ circuit.

and the gate level model are the same. In order for a gate level model to allow a circuit to verify at a higher clock speed than an abstracted model there need to be instances of fast gates in one stage feeding slow gates in another block in the next stage. Such instances do not occur in this example.

9.5 Summary

Our results show that the POSET algorithm when applied to TEL structures can dramatically improve the efficiency of timing verification allowing larger, more concurrent timed systems to be verified. It does so without eliminating parts of the state space, so it does not limit the properties that can be verified. Due to the efficiency of the algorithm and the flexibility of TEL structures, ATACS is very effective for the verification of both both synchronous and asynchronous circuits. Since ATACS is designed for asynchronous circuits, it can be used to verify many different circuit styles by varying the constraints that are checked. When circuit-level timing specifications can be verified, less margin is necessary in each circuit to ensure that the circuit works correctly, which can result in higher performance. ATACS does a complete state space exploration. Therefore, its complexity is exponential and it is not practical to verify large circuits at the gate level. However, for most circuits, a higher level of abstraction is sufficient to verify that the circuit can run at the desired speed. If this is not the case, it is possible to locally specify

more detail on paths that fail without causing state explosion.

9.6 Appendix - Reproducibility

All results are run using the version of ATACS checked in to the ATACS CVS tree on `ming.elen.utah.edu` under the tag `wendy_thesis`. All of the specification used in this chapter are checked into the CVS tree under the directory `examples`, in either the `csp`, `er`, or `tel` directory. The following table describes the location of the specification for every example in this section, the switches used to produce each result, and the machine that the results are produced on.

Example	Machine	Location	Column	Switches
n-bit counter	ming	csp/cnt n _synch.csp	geom	geometric
			geom+All	geometric, subsets, supersets, interleav
			PO	posets
			sub/sup	posets, subsets supersets
			inter	posets, interleav
			all	posets, subsets, supersets, interleav
n-stage FIFO	ming	csp/lapb n sv.csp	geom	geometric
			geom+All	geometric, subsets, supersets, interleav
			PO	posets
			sub/sup	posets, subsets supersets
			inter	posets, interleav
			all	posets, subsets, supersets, interleav
n-stage select	ming	er/selectorn.csp	geom	geometric, orbmatch
			g+A	geometric, subsets, supersets, interleav, orbmatch
			PO	posets, orbmatch
			sub/sup	posets, subsets, supersets, orbmatch
			inter	posets, interleav, orbmatch
			all	posets, subsets, orbmatch, supersets interleav, orbmatch
			-M	posets, subsets, supersets, interleav
			app.	poapprox, subsets, supersets, interleav, orbmatch

Table 9.6. Location of examples.

Example	Machine	Location	Column	Switches
n-stage tag	ming	er/tag n .csp	geom	geometric, orbmatch
			g+A	geometric, subsets, supersets, interleav, orbmatch
			PO	posets, orbmatch
			sub/sup	posets, subsets, supersets, orbmatch
			inter	posets, interleav, orbmatch
			all	posets, subsets, orbmatch, supersets interleav, orbmatch
			-M	posets, subsets, supersets, interleav
			app.	poapprox, subsets, supersets, interleav, orbmatch
level tag	ming	csp/tag_level.csp	geom	geometric, orbmatch, postproc
			g+A	geometric, subsets, postproc supersets, interleav, orbmatch, postproc
			PO	posets, orbmatch, postproc
			sub/sup	posets, subsets, supersets, orbmatch, postproc
			inter	posets, interleav, orbmatch
			all	posets, subsets, postproc orbmatch, supersets interleav, orbmatch, postproc
			-M	posets, subsets, supersets, interleav
			app.	poapprox, subsets, supersets, interleav, orbmatch, postproc
alpha	ching	csp/alpha.csp	n/a	posets, subsets, supersets
beta	ching	csp/beta.csp	n/a	posets, subsets, supersets
n-stage stari	ching	er/stari_old n .er	n/a	posets, subsets, supersets

Table 9.7. Location of examples - continued.

Example	Machine	Location	Column	Switches
arbiter	ming	tel/arbiter.er	explicit	posets, subsets, supersets
			implicit	posets, subsets supersets, bdd
big arbiter	ming	tel/arbiter_big.er	explicit	posets, subsets, supersets
			implicit	posets, subsets supersets, bdd
mle	ming	tel/mle.tel	n/a	disabling, posets
pla	ming	tel/pla.tel	n/a	posets
comparator	ming	domino_compare.tel	n/a	posets, subsets, supersets
mfxu	ming	tel/mfxu.tel	n/a	posets, subsets, supersets
clz	ming	tel/clz.tel	n/a	posets, subsets supersets

Table 9.8. Location of examples - continued.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

What does not kill me makes me stronger.
- Friedrich Nietzsche

10.1 Summary

The results from the previous chapter show that the algorithms presented in this thesis significantly improve the efficiency of timed state space exploration, allowing larger, more concurrent timed systems to be synthesized and verified. These improvements come from many sources. The first is in the improvement in the specification method. The main advantage of the TEL structure formalism is that it conforms much more closely to gate level circuits than purely event based formalism. This makes circuit specifications easier to construct by hand and also easier to generate automatically. It also results in more concise circuit specifications, which reduces the memory and runtime necessary to do state space exploration. The next improvement comes from the POSET algorithm. The POSET algorithm is the key to improving performance on highly concurrent specifications. The POSET algorithm computes a single geometric region for many firing sequences that differ in the firing order of concurrent events and dramatically reduces the number of regions generated. Although the POSET algorithm is first presented by Rokicki in [58], it is very limited there. This thesis presents a version of the POSET algorithm which works on a very broad class of specifications and it presents theory supporting the POSET algorithm which is missing in [58]. Additional improvements come from the optimizations. As Chapter 9 shows, optimizations, especially the optimization which eliminates redundant rule firing interleavings, have a huge impact on performance. Also, for situations where memory is the limiting factor, the MTBDD optimization is presented which decreases memory consumption by as much as an order of magnitude. In order to better specify verification properties, this thesis presents constraint rules. Constraint rules allow for the verification of interesting concrete time properties without adding significant overhead

that impacts the performance of the algorithm. Finally, the flexibility of the methodology is shown by applying it to timed synchronous circuits. This shows that asynchronous techniques can be useful to synchronous designers, even if they never believe that they are designing asynchronous circuits.

10.2 Future Work

Although we believe this thesis makes a significant contribution to the area of timed circuit design automation, there are many areas which it leaves unexplored. The work presented here can be extended in many directions to improve the specification method, optimize the algorithms, and increase the number and size of applications.

10.2.1 Specification

The examples of the tag unit and the IBM circuits show that the TEL structure specification method conforms well to actual logic gates. However additional expressiveness can be added to make the representation of the circuit more precise. Currently the delay range on a rule is fixed. In physical circuits, the time it takes for a wire to switch depends on which transistor in the stack is the last one to activate. If this trigger transistor is directly connected to the switching wire, the rise or fall time of the wire is less than if the trigger transistor is at the bottom of the stack, connected to power or ground. Using the current TEL structure specification, the variance can be partially modeled by using multiple behavioral rules to enable an event. However, it is difficult to model gates this way. Gates are most easily modeled using level expressions. When level expressions are used, the variance in delay due to different trigger transistors is modeled by setting delay ranges on each rule that are large enough to represent all possible trigger transistors. In order to make this model more precise, we plan to modify the TEL structure formalism in a way that allows a separate delay range to be specified for each possible causal event for a rule.

If each signal transition has a single trigger transistor, this extension is sufficient to model trigger signal dependent delay. However, delay ranges also vary depending on the number of transistors in the driving stack that are switching at the same time. If many driving transistors switch at once, the delay on the output wire is more than if there is only a single trigger transistor. Modeling this behavior is somewhat more difficult. The model would need to contain a delay range for each combination of causal events for each rule with a level expression and the algorithm must be modified to determine

which combinations of trigger transistors can simultaneously switch. This is a significant increase in complexity. In the future we plan to determine if the increased accuracy justifies the increase in complexity.

Improvements can also be made to the verification process. Although we believe that constraint rules are an effective way to specify concrete verification properties, there is also value in the ability to specify and verify properties in a formal timed logic. A logic could be implemented in two ways. If it is possible to express the properties of a logic using constraint rules, then a translation algorithm which converts timed logic formulas into constraints is likely to be the best approach. If this is not possible, then substantial modifications to the POSET algorithm are necessary in order to check logic formulas.

10.2.2 Algorithms

There are a number of ways the POSET algorithm can be improved and extended to allow for synthesis and verification of larger designs. The first is the application of partial orders to verification. The current POSET algorithm finds the entire untimed state space, which is necessary for synthesis. However, in verification many states are often not relevant to the properties being verified. In a specification with a large untimed state space, such as the arbiter, partial orders can significantly reduce state space size by eliminating irrelevant untimed states. In the future, we plan to combine Valmari and Godefroid's partial order approach [67, 31] with the POSET algorithm to explore the potential improvement.

Another possible algorithmic extension involves arbitrary boolean expressions. Although TEL structures allow arbitrary boolean expressions, the POSET algorithm cannot currently analyze TEL structures containing them. Although specifications can be transformed to avoid arbitrary boolean expressions, it is more efficient to analyze them directly if possible. In the future, we plan to extend the POSET algorithm to operate on TEL structures with arbitrary boolean expressions.

The next algorithmic improvement concerns the use of implicit methods. Currently MTBDDs are used as a storage mechanism for the geometric regions to improve memory performance. The first step in extending their use is to implement the stack using MTBDDs. However, this is also simply the use of MTBDDs as a memory optimizing data structure. When implicit methods are used for state space exploration, they are typically used not only to represent the states that have been found, but also to represent a transition function that controls the generation of new states. The transition function

is applied repeatedly to the initial state until a fixed point is reached. This is usually a much faster process than enumerating the states explicitly. The POSET algorithm is very complex, and therefore it may be unrealistic to express it in a single transition function. However, if this can be done, it could eliminate the runtime penalty incurred by the current BDD approach.

The final algorithmic improvement is perhaps the most significant. State space exploration is a fundamentally exponential problem. Although better algorithms increase the size of circuits that can be synthesized and verified, this size will always be relatively small. In order for the technique to scale to industrial size problems, abstraction is needed. The larger IBM examples from Chapter 9 show the potential of abstraction. Gate level models are beyond the capabilities of the algorithm but abstracted models verify relatively quickly. The abstraction in Chapter 9 is done by hand. This is a time consuming and error-prone process. Automated abstraction is needed in order for the timed circuit design methodology to be used extensively by circuit designers. When circuits can be automatically abstracted, it is then possible to create a system for hierarchical verification which scales to very large designs.

10.2.3 Applications

Additional work is also needed in the application of the algorithms presented here. Chapter 9 shows that asynchronous techniques can be applied to synchronous circuits. Since asynchronous algorithms are very general they can be applied directly to a circuit which uses any type of timing assumptions. However, synchronous circuits make a very specific timing assumption by using a clock. Timing analysis techniques designed for synchronous circuits rely too much on this assumption and therefore are not easily extensible to new circuit styles. Asynchronous algorithms ignore it completely and therefore have much worse performance. In the future, we plan to explore how the synchronous assumptions can be integrated more tightly with the asynchronous techniques to improve algorithm performance on timed synchronous circuits.

Although this thesis concentrates on the application of the POSET algorithm to timed circuits, that is only one potential application. There are many problems which can be modeled as timed concurrent systems, such as real-time distributed systems. Although asynchronous circuit researchers and real-time researchers are often working on similar problems, there is little communication. We plan to attempt to remedy this by applying

the work presented here to a broader class of concurrent systems and by looking for work in real-time systems that can be applied to asynchronous circuit design.

The last application is really an extension to the work on the guTS processor shown in Chapter 9. In order to really test a CAD methodology it must be used on a real, industrial scale design in progress. The industrial circuits verified by **ATACS** in this thesis were already known to work at the time of verification. The verification process only fails when these circuits are specified incorrectly. The final measure of a verification technique is how quickly it finds bugs and whether it can find them earlier in the design cycle than other methods such as simulation. This thesis shows that the algorithms presented here can be used to verify circuits, but it does not show if they can increase designer productivity by finding bugs faster. In the future we plan to apply the algorithms developed here to a large, industrial design and build on the knowledge that is gained from the experience.

REFERENCES

- [1] ALUR, R. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, August 1991.
- [2] ALUR, R., COURCOUBETIS, C., AND DILL, D. Model-checking in dense real-time. *Information and Computation* 104, 1 (May 1992).
- [3] ALUR, R., COURCOUBETIS, C., DILL, D., HALBWACHS, N., AND WONG-TOI, H. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the Real-Time Systems Symposium (1992)*, IEEE Computer Society Press, pp. 157–166.
- [4] ALUR, R., FEDER, T., AND HENZINGER, T. The benefits of relaxing punctuality. *Journal of the ACM* 43, 1 (Jan 1996), 116–146.
- [5] ALUR, R., AND HENZINGER, T. A really temporal logic. *Journal of the ACM* 41, 1 (Jan 1994), 181–203.
- [6] ALUR, R., AND KURSHAN, R. P. Timing analysis in cospan. In *Hybrid Systems III (1996)*, Springer-Verlag.
- [7] BELLUOMINI, W., AND MYERS, C. Verification of timed systems using POSETs. In *International Conference on Computer Aided Verification (1998)*, Springer-Verlag.
- [8] BELLUOMINI, W., AND MYERS, C. J. Timed event/level structures. In collection of papers from TAU'97.
- [9] 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, pp. 88–100.
- [10] BELLUOMINI, W., MYERS, C. J., AND HOFSTEE, H. P. Verification of delayed-reset domino circuits using ATACS. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (Apr. 1999)*, pp. 3–12.
- [11] BENGTTSSON, J., JONSSON, B., LILIUS, J., AND YI, W. Partial order reductions for timed systems. In *International Conference on Concurrency Theory (September 1998)*.
- [12] 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).

- [13] 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).
- [14] BROWNE, M. C., CLARKE, E. M., DILL, D. L., AND MISHRA, B. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers* 35, 12 (Dec. 1986), 1035–1044.
- [15] BRUNVAND, E., AND SPROULL, R. F. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer-Aided Design, ICCAD-1989* (1989), IEEE Computer Society Press.
- [16] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35, 8 (Aug. 1986), 677–691.
- [17] BURCH, J. R. Combining ctl, trace theory and timing models. In *Proceedings of the First Workshop on Automatic Verification Methods for Finite State Systems* (1989).
- [18] BURCH, J. R. Modeling timing assumptions with trace theory. In *ICCD* (1989).
- [19] BURCH, J. R. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.
- [20] BURNS, S. M. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.
- [21] CHAPPELL, T. I., CHAPPELL, B. A., SCHUSTER, S. E., ALLAN, J., KLEPNER, S., JOSHI, R., AND FRANCH, R. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE Journal of Solid-State Circuits* 26, 11 (Nov. 1991), 1577–1585.
- [22] CHU, T.-A. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.
- [23] CLARKE, E., EMERSON, E., AND SISTLA, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (Feb. 1986), 244–263.
- [24] CLARKE, E., FUJITA, M., AND ZHAO, X. Application of multi-terminal binary decision diagrams. Tech. Rep. CMU-CS-95-160, Carnegie-Mellon University, 1995.
- [25] 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.
- [26] CORTADELLA, J., KISHINEVSKY, M., LAVAGNO, L., AND YAKOVLEV, A. Synthesizing petri nets from state-based models. In *International Conference on Computer-Aided Design* (November 1995).
- [27] DILL, D. L. Trace theory for automatic hierarchical verification of speed-independent circuits. In *Advanced Research in VLSI* (1988), J. Allen and F. T. Leighton, Eds., MIT Press, pp. 51–65.

- [28] 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).
- [29] DILL, D. L. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.
- [30] DILL, D. L., AND CLARKE, E. M. Automatic verification of asynchronous circuits using temporal logic. In *1985 Chapel Hill Conference on VLSI* (1985), H. Fuchs, Ed., Computer Science Press, pp. 127–143.
- [31] GODEFROID, P. Using partial orders to improve automatic verification methods. In *International Conference on Computer-Aided Verification* (June 1990), pp. 176–185.
- [32] GREENSTREET, M., AND ONO-TESFAYE, T. A fast, asP*, RGD arbiter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 173–185.
- [33] GREENSTREET, M. R. Stari: Skew tolerant communication. unpublished manuscript, 1997.
- [34] GREENSTREET, M. R. *STARI: A Technique for High-Bandwidth Communication*. PhD thesis, Princeton University, 1993.
- [35] HOFSTEE, H. P., DHONG, S. H., MELTZER, D., NOWKA, K. J., SILBERMAN, J. A., BURNS, J. L., POSLUSZNY, S. D., AND TAKAHASHI, O. Designing for a Gigahertz. *IEEE MICRO* (May-June 1998).
- [36] HULGAARD, H. *Timing Analysis and Verification of Timed Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Washington, 1995.
- [37] HULGAARD, H., AND BURNS, S. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (November 1994), pp. 2–11.
- [38] HULGAARD, H., BURNS, S., AMON, T., AND BORRIELLO, G. Practical applications of an efficient time separation of events algorithm. In *ICCAD* (1993).
- [39] KOGGE, P., AND STONE, H. A parallel algorithm for the efficient solution of a general class of recurrence relations. *IEEE Transactions on Computers* (Aug. 1973), 786–793.
- [40] KUKIMOTO, Y., AND BRAYTON, R. K. Delay characterization of combinational modules. In *International Workshop on Logic Synthesis* (1998).
- [41] LAVAGNO, L., KEUTZER, K., AND SANGIOVANNI-VINCENTELLI, A. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference* (1991), IEEE Computer Society Press, pp. 302–308.
- [42] LEWIS, H. R. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Tech. rep., Harvard University, July 1989.

- [43] LIN, K.-J., AND LIN, C.-S. Automatic synthesis of asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference (1991)*, IEEE Computer Society Press, pp. 296–301.
- [44] MARTIN, A. J. Programming in VLSI: From communicating processes to delay-insensitive circuits. In *Developments in Concurrency and Communication (1990)*, C. A. R. Hoare, Ed., UT Year of Programming Series, Addison-Wesley, pp. 1–64.
- [45] McMILLAN, K. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. International Workshop on Computer Aided Verification (1992)*, G. v. Bochman and D. K. Probst, Eds., vol. 663 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 164–177.
- [46] McMILLAN, K., AND DILL, D. L. Algorithms for interface timing verification. In *International Conference on Computer Design, ICCD-1992 (1992)*, IEEE Computer Society Press.
- [47] 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.
- [48] 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, pp. 279–289.
- [49] MOON, C. W., STEPHAN, P. R., AND BRAYTON, R. K. Specification, synthesis and verification of hazard-free asynchronous circuits. *Journal of VLSI Signal Processing* 7, 1/2 (Feb. 1994), 85–100.
- [50] MYERS, C. J. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.
- [51] MYERS, C. J., AND MENG, T. H.-Y. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems* 1, 2 (June 1993), 106–119.
- [52] MYERS, C. J., ROKICKI, T. G., AND MENG, T. H.-Y. Automatic synthesis of gate-level timed circuits with choice. In *16th Conference on Advanced Research in VLSI (1995)*, IEEE Computer Society Press, pp. 42–58.
- [53] MYERS, C. J., ROKICKI, T. G., AND MENG, T. H.-Y. Poset timing and its application to the synthesis and verification of gate-level timed circuit. *IEEE Transactions on CAD* 18, 4 (June 1999), 769–786.
- [54] NOWICK, S. M., YUN, K. Y., AND DILL, D. L. Practical asynchronous controller design. In *Proc. International Conf. Computer Design (ICCD) (Oct. 1992)*, IEEE Computer Society Press, pp. 341–345.
- [55] NOWKA, K., GALAMBOS, T., AND DHONG, S. Circuit design techniques for a Gigahertz integer microprocessor. In *International Conference on Computer Design (Oct. 1998)*.

- [56] PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th IEEE symposium on the Foundations of Computer Science* (1982), pp. 195–220.
- [57] POSLUSZNY, S., AND ET AL. Design methodology for a 1.0 GHz microprocessor. In *International Conference on Computer Design* (1998).
- [58] ROKICKI, T. G. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.
- [59] ROKICKI, T. G., AND MYERS, C. J. Automatic verification of timed circuits. In *International Conference on Computer-Aided Verification* (1994), Springer-Verlag, pp. 468–480.
- [60] ROTEM, S., STEVENS, K., GINOSAR, R., BEEREL, P., MYERS, C., YUN, K., KOL, R., DIKE, C., RONCKEN, M., AND AGAPIEV, B. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1999), pp. 60–70.
- [61] SEMENOV, A., AND YAKOVLEV, A. Verification of asynchronous circuits using time Petri-net unfolding. In *Proc. ACM/IEEE Design Automation Conference* (1996), pp. 59–63.
- [62] SILBERMAN, J., AND ET AL. A 1.0 GHz single issue 64-bit PowerPC integer processor. *IEEE Journal of Solid-State Circuits* (Nov. 1998), accepted for publication.
- [63] SUBRAHMANYAM, P. What’s in a timing discipline? considerations in the specification and synthesis of systems with interacting asynchronous and synchronous components. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects* (1990), Springer-Verlag.
- [64] TASIRAN, S., AND BRAYTON, R. K. Stari: A case study in compositional and hierarchical timing verification. In *Proc. International Conference on Computer Aided Verification* (1997).
- [65] THACKER, R. A. Implicit methods for timed circuit synthesis. Master’s thesis, University of Utah, 1998.
- [66] THACKER, R. A., BELLUOMINI, W., AND MYERS, C. J. Timed circuit synthesis using implicit methods. In *International Conference on VLSI Design* (1999), pp. 181–188.
- [67] VALMARI, A. A stubborn attack on state explosion. In *International Conference on Computer-Aided Verification* (June 1990), pp. 176–185.
- [68] VAN BERKEL, C., AND SAEIJS, R. Compilation of communicating processes into delay-insensitive circuits. In *International Conference on Computer Design, ICCD-1988* (1988), IEEE Computer Society Press.
- [69] VANBEKBERGEN, P., CATTHOOR, F., GOOSSENS, G., AND MAN, H. D. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (1990), IEEE Computer

Society Press, pp. 184–187.

- [70] VANBEKBERGEN, P., GOOSSENS, G., CATTHOOR, F., AND MAN, H. J. D. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. *IEEE Transactions on Computer-Aided Design* 11, 11 (Nov. 1992), 1426–1438.
- [71] VERLIND, E., DE JONG, G., AND LIN, B. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference* (1996).
- [72] WINSKEL, G. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Noordwijkerhout, Norway (June 1988).
- [73] YONEDA, T., SHIBAYAMA, A., SCHLINGLOFF, B., AND CLARKE, E. M. Efficient verification of parallel real-time systems. In *Computer Aided Verification* (1993), C. Courcoubetis, Ed., Springer-Verlag, pp. 321–332.
- [74] 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.
- [75] YUN, K. Y., DILL, D. L., AND NOWICK, S. M. Practical generalizations of asynchronous state machines. In *Proc. European Conference on Design Automation (EDAC)* (Feb. 1993), IEEE Computer Society Press, pp. 525–530.
- [76] ZHENG, H. Specification and compilation of timed systems. Master's thesis, University of Utah, 1998.