COMPUTER-AIDED SYNTHESIS AND VERIFICATION OF GATE-LEVEL TIMED CIRCUITS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Christopher John Myers

October 1995

© Copyright 1995
by
Christopher John Myers

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Teresa H.-Y. Meng
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Adam P. Arkin

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

In recent years, there has been a resurgence of interest in the design of *asynchronous circuits* due to their ability to eliminate clock skew problems, achieve average case performance, adapt to processing and environmental variations, provide component modularity, and lower system power requirements. Traditional academic asynchronous designs methods use unbounded delay assumptions, resulting in circuits that are verifiable, but ignore timing for simplicity, leading to unnecessarily conservative designs. In industry, however, timing is critical to reduce both chip area and circuit delay. Due to a lack of formal methods that handle timing information correctly, circuits with timing constraints usually require extensive simulation to gain confidence in the design.

This thesis bridges this gap by introducing *timed circuits* in which explicit timing information is incorporated into the specification and utilized throughout the design procedure to optimize the implementation. Our timed circuits are more efficient than those produced using untimed methods and more reliable than those produced using *ad hoc* design techniques. Timing analysis, however, often introduces substantial complexity into the design procedure, and has hitherto either been avoided, simplified, or considered only after synthesis. In this thesis, we describe an exact and efficient timing analysis algorithm, and its application to the automatic synthesis and verification of gate-level timed circuits. Our synthesis procedure generates hazard-free timed circuits and maps the resulting implementations to practical, semi-custom gate libraries. The resulting implementations are up to 40 percent smaller and 50 percent faster than previous asynchronous designs. We also demonstrate that our timed designs can be smaller and faster than their synchronous counterparts. After back-annotating the synthesized circuits, our verification procedure checks that all circuits satisfy their specifications. This procedure has also been applied to a wide collection of highly concurrent timed circuits that could not previously be verified.

Acknowledgments

I am indebted to my adviser, Professor Teresa Meng, for suggesting this research topic to me and giving me support and patience as I struggled with it. Professor David Dill, my associate adviser, provided substantial technical support, and I would like to thank him for many very enlightening discussions about timed circuits and many other matters. I am grateful to Dr. Adam Arkin for serving as my third reader and showing me a wider scope of application for my research. Professor Giovanni DeMicheli is also gratefully acknowledged for his advice on my work. Finally, I would like to thank Professor Greg Kovacs and Professor Kunle Olukotun for serving on my orals committee.

I have been very fortunate in my years at Caltech and Stanford to get the opportunity to work with many brilliant people. I would like to thank Alain Martin for introducing me to the world of asynchronous design. Peter Beerel has been my colleague, officemate, and friend since I arrived at Stanford. I greatly appreciate his comments and criticizisms which significantly improved the quality of this work. In particular, his collaboration on the synthesis and technology mapping chapters is gratefully acknowledged. I was also very fortunate to be able to collaborate with Tom Rokicki on the timing analysis and verification chapters. I'm indebted to him for helping me deal with a world filled with choices. I would also like to thank all the past members of the Stanford asynchronous group (Jerry Burch, Bill Coates, Al Davis, Jeremy Gunawardena, Steve Nowick, Polly Siegel, and Ken Yun) for providing a stimulating working environment. Finally, I would like to thank Steve Burns and Gaetano Borriello for their advice and support through the years.

Numerous individuals provided invaluable moral support. In particular, I would like to thank Janet Lai for her friendship. I'm grateful to Lilian Betters for her administrative help throughout the years, especially during my job search. I would also like to thank the many other friends who gave me wonderful distractions from work (Francis Chong, Korhan Gurkan, Jeff Jones, Joe Lauer, John Lazzaro, Garland Lee, Jared Levy, Amit Mehra, Misha

Samoilov, Craig Sosin, Emily Wen, and Su-lin Wu), the other members of Teresa's group (Navin Chaddha, Ben Gordon, Andy Hung, Won Namgoong, Clem Portmann, Wee-chiew Tan, Tony Todesco, and Ely Tsern), and my parents and family for their encouragement and support.

This work was supported by an NSF fellowship, the Semiconductor Research Corporation contract 93-DJ-205, a grant from the NSF PYI Program, the Office of Navel Research contract N00014-89-J-3036, and the Advanced Research Projects Agency contract DABT63-91-K-0002. Finally, much of this thesis was reworked at Intel in Israel during the summer of 1995 while consulting with Shai Rotem whom I am grateful to for allowing me the opportunity to apply my work.

Contents

1 Introduction **1**

1.1 Asynchronous Circuit Design . . . 2

1.1.1 Delay-Insensitive Circuits . . . 4

1.1.2 Quasi-Delay Insensitive and Speed-Independent Circuits . . . 5

1.1.3 Fundamental-Mode Circuits . . . 5

1.1.4 Timed Circuits . . . 6

1.2 Contributions . . . 7

1.3 Thesis Overview . . . 8

2 Timed Specifications **10**

2.1 Timed Handshaking Expansions . . . 10

2.1.1 Modules, Signal Declarations, and Processes . . . 11

2.1.2 Basic Commands and Their Composition . . . 12

2.1.3 Guarded Commands . . . 12

2.1.4 Example . . . 14

2.2 Timed Event-Rule Structures . . . 18

2.3 Timed Configurations . . . 19

2.4 Interpreting the Specification Language . . . 21

2.4.1 Declarations . . . 21

2.4.2 Composition of Timed Event-Rule Structures . . . 22

2.4.3 Renaming of Timed Event-Rule Structures . . . 23

2.4.4 Interpretation of a Non-Repetitive Process . . . 23

2.4.5 Interpretation of a Repetitive Process . . . 25

2.4.6 Vacuous Events . . . 26

2.4.7 Interpretation of a Module . . . 26

2.4.8 Example 27

3 Timing Analysis **32**

3.1 Constraint graphs 33

3.2 Estimating the Worst-Case Time Difference 36

3.2.1 Worst-Case Time Difference 36

3.2.2 Algorithm to Estimate the Worst-Case Time Difference 37

3.2.3 Proof of Correctness 39

3.2.4 Complexity of the Algorithm 40

3.2.5 Extensions to Find a Better Estimate 40

3.2.6 Termination of the Algorithm 41

3.2.7 Removing Redundant Rules 41

3.3 Orbital Nets 42

3.3.1 Timing Requirements 43

3.3.2 Simultaneous Actions 45

3.3.3 Operational Semantics 46

3.3.4 Transformation from a Timed ER Structure to an Orbital Net . . . 47

3.3.5 Satisfying the Single Behavior Place Requirement 48

3.4 Partial Order Timing 50

3.4.1 Geometric Regions 53

3.4.2 State Space Exploration with Geometric Timing 54

3.4.3 Performance of Geometric Timing 54

3.4.4 Concurrency, Causality, and Posets 55

3.4.5 State Space Exploration with Partial Order Timing 57

3.4.6 Efficiency Considerations 58

3.5 Finding the Reduced State Graph 58

4 Synthesis **65**

4.1 Sum-of-Products Implementation 66

4.2 Generalized C-Implementation 67

4.3 Standard C-Implementation 69

4.3.1 Excitation Regions and Quiescent States 69

4.3.2 Correct Covers 70

4.4 Finding Enabled Cubes and Trigger Cubes 71

4.5 Finding an Optimal Correct Cover 72
4.6 Synthesis Results 76

5 Technology Mapping **80**
5.1 Gate Libraries 81
5.2 Decomposition 82
5.2.1 Searching the Decomposition Space 83
5.2.2 Decomposition Through Resynthesis 84
5.2.3 Multi-level Decompositions 88
5.3 Example 89
5.4 Technology Mapping Results 91

6 Design Examples **94**
6.1 MMU Controller 94
6.1.1 The Memory Data Load Cycle 95
6.1.2 Complete MMU 99
6.2 DRAM Controller 102
6.3 Two-bit Synchronous Counter 106

7 Verification **109**
7.1 Behavioral Semantics 110
7.2 Generating the Orbital Net Representations 111
7.3 Reporting Failures 112
7.4 Verification Results 113

8 Conclusions **116**
8.1 Summary 116
8.2 Future Work 117
8.2.1 Specification 117
8.2.2 Compilation 117
8.2.3 Technology Mapping and Module Generation 118
8.2.4 Verification 118
8.2.5 Asynchronous Datapaths 119
8.2.6 Interfacing with Synchronous Designs 119

List of Tables

1 Enabled cubes and trigger cubes for the SEL. 72

2 CC table for $(out2_o \downarrow, 0)$ from the SEL. 74

3 CC table for $(out2_o \downarrow, 0)$ from the SEL after removing dominating columns. 75

4 Synthesis results. 77

5 Technology mapping results. 92

6 Production rules for speed-independent and timed circuits for the MDl cycle. 98

7 Verification results. 114

List of Figures

1 Modules, signal declarations, and processes. 12
2 Basic commands and their composition. 12
3 Guarded commands. 13
4 (a) CSP specification and (b) block diagram for a port selector (SEL). . . . 14
5 Part of the timed HSE specification for the SEL. 17
6 Reshuffling of the *selctrl* process. 17
7 Complete BNF description for the timed HSE specification language. 29
8 Part of the timed HSE specification for the SEL. 30
9 Format for a timed ER structure. 30
10 Timed ER structure for the *sel* process from the SEL. 31
11 Timed HSE specification for a SCSI protocol controller. 34
12 Cyclic constraint graph for a SCSI protocol controller. 34
13 Part of the acyclic constraint graph for the SCSI protocol controller. 35
14 Algorithm to find an estimate of the worst-case time difference in a cyclic graph. 38
15 Algorithm to find a time difference in an acyclic graph. 38
16 Algorithm to find a maximum time difference in an acyclic graph. 39
17 Algorithm to find redundant rules. 42
18 (a)A D-type flip-flop; (b) its timing requirements represented using a timing diagram; (c) its timing requirements represented using an orbital net. . . . 44
19 (a) AND gate with inputs a and b, and output d; (b) orbital net for its functional behavior; (c) delay buffer with input c, output d, and delay of $\langle 2, 4\rangle$. 46
20 Algorithm to transform a timed ER structure to an orbital net. 49
21 (a) Orbital net for the *sel* process from the SEL; (b) part of the orbital net after composition with the other processes. 50

22 (a) Fragment of the orbital net that violates the single behavior place requirement; (b) graphical representation of the desired timing behavior. . . . 51
23 (a) Orbital net that satisfies the single behavior place requirement; graphical representation of the timing behavior of c_0 (b) and c_1 (c). 51
24 (a) Fragment of an orbital net with a behavior place that has multiple transitions in its postset; (b) part of the transformed orbital net that satisfies the single behavior place requirement. 51
25 (a) Unit-cube, (b) discrete, and (c) geometric representations of the timed state space. 52
26 The adverse example `adv4x40` with $n = 4$ and $k = 40$. 55
27 Geometric regions from the adverse example. 56
28 One poset from the adverse example. 56
29 Algorithm to find the reduced state graph. 60
30 Algorithm to check if an event is slow. 61
31 (a) SG and (b) RSG for the SCSI protocol controller. 62
32 Redundant rules from the SCSI protocol controller. 63
33 Reduced state graph for the SCSI protocol controller. 64
34 A hazardous sum-of-products implementation of $out2_o$ from the SEL. 67
35 (a) The generalized C-element configuration with (b) weak-feedback and (c) fully-static CMOS implementations. 68
36 A hazardous gate-level implementation of $out2_o$ from the SEL. 68
37 The standard C-implementation. 69
38 Gate-level (a) timed and (b) speed-independent circuits for the SEL. 76
39 Enabled cubes and trigger cubes from the SCSI protocol controller. 78
40 CC table from the SCSI protocol controller. 79
41 Production rules from the SCSI protocol controller. 79
42 (a) Part of the orbital net for the *tsbm*, (b) a standard C-implementation, and (c) a generalized C-implementation of the signal $DReq_o$. 83
43 (a) Part of the orbital net for a decomposition using a trigger signal, and (b) corresponding generalized C-implementation. 86
44 (a) Part of the orbital net for a decomposition using a context signal, and (b) corresponding generalized C-implementation. 87

45 (a) Part of the orbital net for a multi-level decomposition, and (b) corresponding generalized C-element implementation of $DReq_o$ with a maximum fanin of two. 88
46 (a) Part of the orbital net for the *SELopt*, and (b) part of the orbital net after a decomposition of the signal sel_o. 90
47 The gate-level timed circuit implementation of the *SELopt* (a) before decomposition; after decomposition to (b) 3-input gates and (c) 2-input gates. . . 91
48 Block diagram for the MDl cycle of the MMU controller. 95
49 The cyclic constraint graph for the unoptimized MDl cycle. 96
50 The cyclic constraint graph for the optimized MDl cycle. 97
51 The cyclic constraint graph for the persistent MDl cycle. 97
52 (a) Timed and (b) speed-independent implementations for the MDl cycle. . 99
53 Part of the timed HSE specification for the complete MMU controller. . . . 100
54 Gate-level (a) timed and (b) speed-independent circuits for the MMU controller. 101
55 Block diagram for a DRAM interface. 103
56 The burst-mode specification for the DRAM controller. 103
57 Part of the timed HSE specification of the DRAM controller. 104
58 Overall implementation of the DRAM controller. 104
59 Complex-gate implementation of the *cas* signal for the DRAM controller. . 105
60 (a) Timed and (b) synchronous circuits for a DRAM controller. 105
61 The cyclic constraint graph specification for a two-bit synchronous counter: (a) initial specification and (b) final specification. 106
62 Complex-gate implementation of a two-bit synchronous counter. 107
63 Implementation of a two-bit synchronous counter derived using `SIS`. 107
64 (a) AND gate with inputs a and b, and output d; (b) orbital net for functional behavior; (c) delay buffer with input c, output d, and delay of $\langle 2, 4 \rangle$. 110
65 Part of the specification orbital net for the SEL. 111
66 Implementation of the SEL which fails verification. 112
67 Seitz queue element. 113

Chapter 1

Introduction

all pain disappears it's the nature of my circuitry
—nine inch nails
I must govern the clock, not be governed by it.
—Golda Meir

There has been a recent resurgence of interest in the design of *asynchronous circuits* due to their potential to provide more robust, higher performance, and lower power implementations. Unfortunately, these advantages have not yet been fully realized as current asynchronous design methodologies either produce inefficient or unreliable designs. Traditional academic asynchronous design methodologies use unbounded delay assumptions, resulting in circuits that are verifiable, but ignore timing for simplicity, leading to unnecessarily conservative designs. In industry, however, timing is critical to reduce both chip area and circuit delay. Due to the lack of formal methods to handle timing information correctly, circuits with timing constraints usually require extensive simulation to gain confidence in the design. Simulation, however, is not perfect so unreliable designs can be produced. This fact has proven to be a major stumbling block to the widespread acceptance of asynchronous circuits within industry.

This thesis bridges this gap between academia and industry by introducing *timed circuits* in which explicit timing information is incorporated into the specification and utilized throughout the design procedure to optimize the implementation. Timed circuits can be significantly smaller and faster than those produced using traditional formal methods, and they are more reliable than those produced using *ad hoc* techniques. The specification

of timing constraints also facilitates a natural interaction between synchronous and asynchronous circuits.

1.1 Asynchronous Circuit Design

An asynchronous circuit is one in which synchronization is performed without a global clock. Asynchronous circuits have several advantages over their synchronous counterparts including:

1. *Elimination of clock skew problems.* As systems become larger, increasing amounts of design effort is necessary to guarantee minimal skew in the arrival time of the clock signal at different parts of the chip. In the *DEC alpha* microprocessor, nearly a third of the silicon area is required for the clock distribution network [25]. In an asynchronous circuit, skew in synchronization signals can be tolerated, so this extra circuitry is not necessary.

2. *Average-case performance.* In synchronous systems, the performance is dictated by worst-case conditions. The clock period must be set to be long enough to accommodate the slowest operation even though the average delay of the operation is often much shorter. In asynchronous circuits, the speed of the circuit is allowed to change dynamically, so the performance is governed by the average-case delay.

3. *Adaptivity to processing and environmental variations.* The delay of a VLSI circuit can vary significantly over different processing runs, supply voltages, and operating temperatures. For this reason, synchronous designs are simulated over a wide variation of these parameters, and the clock is set so that the majority of chips produced operate correctly under some allowed variations. Due to their adaptive nature, an asynchronous circuit operates correctly under all variations and simply speeds up or slows down, as necessary.

4. *Component modularity.* In an asynchronous system, components can be interfaced without the difficulties associated with synchronizing clocks in a synchronous system. Also, when a new faster component becomes available, it can often be easily inserted into the system without requiring any other changes to the rest of the system resulting in a corresponding improvement in system performance.

5. *Lower system power requirements.* Asynchronous circuits reduce synchronization power by not requiring additional clock drivers and buffers to limit clock skew. They can also automatically power down unused components. In many synchronous applications, more than half of the power is wasted with spurious transitions [63]. Asynchronous circuits have no spurious transitions. Finally, asynchronous circuits can easily be adjusted to make efficient use of a dynamic power supply.

While asynchronous designs have long been used in interface circuits, they are now being considered for the design of high-performance processors [45, 76, 27, 55] and low-power embedded controllers and portable devices [71]. Unfortunately, the advantages of asynchronous circuits have not yet been fully realized for several reasons, including:

1. *Lack of mature computer-aided design tools.* In the past several years, there has been a rapid development of commercial VLSI design tools. These tools, however, are limited to synchronous designs. While many asynchronous design methods have supporting CAD tools, these tools are still in the experimental phase.

2. *Large area overhead for the removal of hazards.* A *hazard* is a spurious signal transition, or glitch. While hazards can be ignored in a synchronous design as they are filtered out by the clock signal, any hazard in an asynchronous design can potentially lead to a malfunction. Therefore, careful design is necessary to avoid hazards in an asynchronous design which often leads to a significant increase in circuit area.

3. *Difficulty in interfacing with existing synchronous designs.* Many asynchronous design methods require the ability to slow down the environment by withholding an acknowledgment. When interfacing with a synchronous design, this is typically not possible.

4. *Necessity for custom design.* As the pace of technology increases, the life spans of products decrease forcing the VLSI industry to often turn to semi-custom components such as standard-cells and gate-arrays to improve time-to-market. However, many asynchronous design procedures require the use of special complex-gates.

5. *Unreliable designs.* In order to get efficient implementations, many asynchronous circuit designers play tricks and make assumptions which must be checked with simulation. Unfortunately, simulation is not perfect so unreliable designs can be produced.

This thesis addresses each of these issues by developing computer-aided design tools that make use of timing throughout the design procedure to produce both efficient and reliable designs that can be mapped to practical gate libraries and interfaced with synchronous designs. This thesis concentrates on the automated design of control circuits, since, due to their regular geometry, datapath modules are usually custom designed to optimize for performance. The various techniques for the design of asynchronous datapaths such as *dual-rail encoding* and *bundled data* are described in [67, 13, 22, 21, 44, 47].

Many other techniques have been proposed for the design of asynchronous control circuits. The rest of this section briefly describes several of these approaches categorized by their timing model. For a more complete description, please see [30].

1.1.1 Delay-Insensitive Circuits

A *delay-insensitive circuit* is one in which its correctness is independent of both gate and wire delays. In [43], Martin proved that when the gate-library is restricted to single-output gates, this class of circuits is severely limited to those which use *Muller C-elements*, an asynchronous memory element, as the only multiple-input gates.

In order to address this problem, many researchers introduce several specially designed multiple-output complex gates. Molnar, et. al. [48] developed a technique which used carefully designed modules which are composed in a delay-insensitive manner. These modules could be either clock-free with internal delay elements to guarantee correctness or locally-clocked *Q-modules*. Other module-based delay-insensitive design procedures exploit higher-level descriptions such as *Occam* [13] used by Brunvand or a *trace-based* language [26] proposed by Ebergen. Automatic compilation techniques are applied to these higher-level descriptions to produce circuit implementations using complex-gate modules.

The major advantage of delay-insensitive designs is modularity. Delay-insensitive modules are easily composed without needing to worry about gate or wire delays. They are very robust and can achieve average-case performance. Unfortunately, they have several serious disadvantages. There can be a large area and delay overhead to achieve delay-insensitivity. It is also impossible for a delay-insensitive design to interface delay-insensitively with a synchronous environment. Finally, the modules required can be large and complex custom designed gates which may be difficult to design reliably.

1.1.2 Quasi-Delay Insensitive and Speed-Independent Circuits

Many methodologies have been proposed for the synthesis of *quasi-delay insensitive* and *speed-independent* circuits in which correctness is independent of gate delays but delays of certain wire forks called *isochronic forks* are negligible. The main difference between quasi-delay insensitive and speed-independent circuits is that all forks in speed-independent circuits must be isochronic.

A quasi-delay insensitive design technique was proposed by Martin [44] which begins from a high-level specification using a modified version of Hoare's *communicating sequential processes* (CSP) [33] which is systematically translated to a circuit implementation. A similar technique proposed by van Berkel [69] automatically compiles a circuit described using a language called *Tangram* to a *handshake circuit* implementation. Several speed-independent design techniques are based on the *signal transition graph* (STG) specification such as the work by Chu [17] and Meng [47]. The work by Chu and Meng, however, often produce circuits that require large complex atomic gates. To address this problem, Beerel et. al. [7] developed constraints to add to the synthesis method to produce implementations using only basic gates such as AND gates, OR gates, and C-elements. Since then there has been some additional work in this area by Lin and Lin [42] and Kondratyev et. al. [37].

The primary advantage of quasi-delay insensitive and speed-independent circuits over delay-insensitive designs is the ability to map the design to basic gates thus allowing semi-custom implementations. However, careful design is necessary to guarantee that the isochronic fork assumption is met. There still can be significant area overhead associated with the need to remove hazards in these circuits. Finally, quasi-delay insensitive and speed-independent circuits cannot be interfaced with synchronous environments.

1.1.3 Fundamental-Mode Circuits

Other design methods use the *fundamental-mode* assumption which takes advantage of timing properties in a limited way by assuming that the environment must wait long enough for the circuit to stabilize before inputs are changed. To achieve this, these techniques must assume that the gate and wire delays are bounded.

Techniques using the fundamental-mode assumption originated with Huffman [34], and were later extended by Unger [68]. Original fundamental-mode techniques allowed only a single input to change at a time. Recently, Davis's group at Hewlett Packard [20] extended

fundamental-mode to allow multiple input change. Nowick developed a locally-clocked method based on this work [57], and Yun introduced a clock-free method [83].

By assuming that the circuit has a bounded delay, it is now possible to construct asynchronous circuits which interface with synchronous environments such as the DRAM controller from [58]. Siegal [65] has also shown that with minor modifications that standard synchronous technology mapping techniques can be applied to map these designs to practical semi-custom gate libraries. However, these circuits may require additional delay elements to guarantee that the fundamental-mode assumption is met, degrading the performance. Also, the fundamental-mode assumption must be guaranteed to hold under all operating conditions, so these circuits may not be able to take full advantage of variations in delay such as those resulting from data-dependencies. Finally, since these methods limit the concurrency within a circuit, they may result in inefficient implementations.

1.1.4 Timed Circuits

Timed circuits are a class of asynchronous circuits that incorporate explicit timing information during some portion of synthesis. This timing information is typically given as bounds on gate, wire, and environment delays. Many of the asynchronous designs done in industry today are timed. That is, their correctness is dependent on meeting certain timing constraints. However, the techniques used for the design of these circuits are typically *ad hoc*, and can result in unreliable designs.

Some systematic techniques exist for the design of timed circuits. Borriello describes in [9] a method which uses timing information in the design of *transducers*, interfaces between synchronous and asynchronous circuits. Lavagno in [38] develops a synthesis technique which uses methods similar to Chu [17] and Meng [47] to get a complex gate implementation which is then mapped to a gate library using synchronous technology mapping techniques. In both of these approaches, timing analysis is applied only after synthesis to verify that hazards do not exist. If hazards are detected, delay elements are added to avoid them, degrading the reliability and performance of the implementation. Beerel et. al. has shown in [7] that the more conservative speed-independent model while resulting in somewhat larger circuits actually produces faster circuits compared with the timed circuits described in [38]. This surprising result can be attributed to the fact that these timed circuits often need to have delay elements added to the critical path to remove hazards.

1.2 Contributions

The major contribution of this thesis is a new automatic method for the synthesis and verification of gate-level timed circuits. The development of a systematic design procedure that incorporates timing information bridges the gap between systematic, untimed asynchronous design methods in academia and *ad hoc*, timed asynchronous design methods in industry. The resulting implementations are both more efficient than previous untimed methods and more reliable than previous timed methods. The specification of timing constraints also facilitates a natural interaction between synchronous and asynchronous circuits.

We outline our contributions in more detail, as follows:

We have proposed a methodology for the specification of timed circuits using a high-level language description. The specification is capable of specifying causality, concurrency, and conditional behavior, or choice, and it is shown to be general enough to specify practical systems. We have also developed procedures for the automatic compilation of this high-level language into a lower-level graphical representation which is conducive to automated timing analysis procedures.

We have developed two efficient timing analysis algorithms. One is a heuristic algorithm used to find the time difference between any two events in a deterministic graphical specification. The other is an exact and efficient method of exploring the timed state space using *geometric regions* and *partial order* information. We have applied these timing analysis algorithms to both the synthesis and verification of timed circuits.

To address synthesis, we have created a complete synthesis procedure from a high-level language to a hazard-free timed circuit. A design strategy and correctness constraints are derived to facilitate the use of semi-custom components. Since the synthesis procedure utilizes the timing information throughout the design procedure to optimize the implementation, extra circuitry is only added to remove hazards that are shown to be able to occur under the given timing constraints. Therefore, our timed circuits can be significantly smaller and faster than those produced using traditional untimed methods. Our synthesis procedure has been fully automated in the CAD tool `ATACS` and applied to several examples. The resulting timed circuit implementations are not only up to 40 percent smaller and 50 percent faster than implementations produced using other asynchronous design methodologies, but also they are can be smaller and faster than their synchronous counterparts.

We have developed an automated procedure for the technology mapping of our timed circuits to practical gate libraries. After using our synthesis procedure to generate a technology-independent timed circuit netlist, the procedure then investigates simultaneous decompositions of all high-fanin gates by adding state variables to the the specification and performing resynthesis. Although multiple decompositions are explored, timing information is utilized to significantly reduce their number. Once all gates are sufficiently decomposed, the netlist can be mapped to the given gate library, taking advantage of any compact complex gates available.

To address verification, we have developed a complete verification procedure capable of checking that the circuit that is built satisfies its original specification. After synthesis, the timed circuit implementation is back-annotated with bounds on the minimum and maximum delay of each gate taken from the given cell-library and verified to satisfy its timed specification. The verification procedure has also been fully automated, and it is shown to be able to rapidly verify larger, more concurrent timed circuits than could previously be verified using traditional techniques.

1.3 Thesis Overview

A circuit is initially described using a high-level specification language. Chapter 2 describes both the formal syntax and semantics of the initial specification language.

In order to perform synthesis or verification it is necessary to determine the reachable state space of the system under consideration. For timed systems, this requires an efficient timing analysis algorithm. Chapter 3 describes two timing analysis algorithms including an exact and efficient timing analysis algorithm used in the subsequent chapters for synthesis, technology mapping, and verification.

Chapter 4 describes the complete synthesis procedure from a high-level language specification to a hazard-free gate-level timed circuit implementation. This chapter first develops correctness constraints at a theoretical level which must be met by the synthesis procedure. Then, it describes the synthesis algorithms in detail.

Chapter 5 describes a procedure to map our timed circuits to practical gate libraries. In particular, this chapter describes a new technique to decompose high-fanin gates. This decomposition technique uses an iterative procedure to guarantee correctness under the given timing constraints.

In order to illustrate the use of the timed circuit design procedure, chapter 6 presents several design examples.

Once the circuit is synthesized, it is back-annotated with delays from the given gate library and verified. Chapter 7 describes the verification procedure.

Finally, chapter 8 gives our conclusions and some ideas for directions of future research.

Chapter 2

Timed Specifications

. . . an event is an action which one can choose to regard as indivisible
—it either has happened or has not according to our description of some process.
This is not to say that an event is indivisible, and without detailed structure,
. . . historians may talk of the event of a battle or the birth of a famous person
—not just single events to the people involved at the time!

—Glynn Winskel

The first step in any design is to specify what is to be built. Many approaches have been taken for the specification of asynchronous circuits. Some approaches use languages such as *communicating sequential processes* (CSP) [44], *Occam* [12], and *Tangram* [70]. Other approaches use graphs such as *I-nets* [48], *signal transition graphs* [17] [47], *change diagrams* [75], *burst-mode state machines* [20] [56] [81], and *state graphs* [7]. While graphs are conducive to automated timing analysis and synthesis algorithms, they are cumbersome for specifying a large system. Languages, however, allow large designs to be specified clearly and concisely. For these reasons, we use a language as the initial specification of our timed designs which is then compiled as described in the next chapter to a graphical representation for timing analysis. This chapter formally defines both the syntax and semantics of our specification language.

2.1 Timed Handshaking Expansions

Our timed circuits are specified using *timed handshaking expansions* (HSE). These specifications are easily derivable, as illustrated in an example later, using techniques similar to those described in [44] from a higher-level description in Martin's version [44] of Hoare's

CSP language [33]. The syntax of the timed HSE language is described in this section. The untimed portion of the timed HSE language is similar in form to Martin's handshaking expansions used in the design of speed-independent asynchronous circuits [44]. Timing is added to the specification by associating a lower and upper bound on the delay of each signal transition in the signal's declaration.

In this section, each language construct is first described informally, and at the end of each subsection the syntax rules of the language constructs are given using an abstract grammar. An abstract grammar is a common way of providing a semi-formal description of a language concisely by ignoring issues such as precedence and ambiguity. The language is defined precisely using a BNF description in an appendix at the end of this chapter. In the abstract grammar notation, the left-hand side and right-hand side are separated by "::=". If there are multiple alternatives on the right-hand side, they are separated by "|". In the syntax rules, boldface words denote keywords, words enclosed in angle brackets "$\langle\ \rangle$" denote language constructs which are described by other syntax rules, italicized words denote language constructs which are only informally described in the text, and "ID" represents an identifier.

2.1.1 Modules, Signal Declarations, and Processes

A module specified in the timed HSE language is composed of two parts: a set of *signal declarations* and a set of concurrent *processes* executing in parallel. The signal declarations are used to specify attributes for each signal wire. Each declaration consists of a type (either **input** or **output**), a signal name, an initial value (either **true** or **false**), and delays associated with transitions on the signal. A delay is given in the form: $\langle l_r, u_r; l_f, u_f \rangle$ where l_r and u_r are the lower and upper bounds on a rising transition and l_f and u_f are the lower and upper bounds on a falling transition. If the fall times are not specified, they are assumed to be equal to the rise times. The lower bounds are nonnegative integers, and the upper bounds are an integer greater than or equal to the lower bound, or ∞. Since real values can be expressed as rationals within any required accuracy, restricting the bounds to be integers does not limit the expressiveness. Since there are only a finite number of timing parameters, if any are rational, we can multiply all of them by the least common denominator. The processes are used to specify the behavior of a module. Each process consists of a set of commands as described in the following two subsections. The parts of the grammar just described are shown in Figure 1.

$$\begin{array}{rcl} \langle\text{module}\rangle & ::= & \textbf{module}\ \text{ID};\langle\text{sigdecl}\rangle\langle\text{process}\rangle\ \textbf{endmodule} \\ \langle\text{sigdecl}\rangle & ::= & \langle\text{sigdecl}\rangle\langle\text{sigdecl}\rangle \mid \textit{type}\ \text{ID} = \{\textit{initial, delay}\ \}; \\ \langle\text{process}\rangle & ::= & \langle\text{process}\rangle\langle\text{process}\rangle \mid \textbf{process}\ \text{ID};\langle\text{cmd}\rangle\ \textbf{endprocess} \end{array}$$

Figure 1: Modules, signal declarations, and processes.

2.1.2 Basic Commands and Their Composition

Each basic command is an *event*. An event specifies when a signal transition can occur. There are two transitions associated with each signal s in a specification, namely, $s \uparrow$ where $\uparrow$ denotes that the signal s is changing from a low to high value, and $s \downarrow$ where $\downarrow$ denotes that the signal s is changing from a high to low value. The language also includes a **skip** event which does nothing and terminates immediately. Commands can be executed either in sequence (denoted $C_1\ ;\ C_2$) or in parallel (denoted $C_1 \parallel C_2$). The constructs just described are shown in Figure 2.

$$\begin{array}{rcl} \langle\text{cmd}\rangle & ::= & \langle\text{cmd}\rangle;\langle\text{cmd}\rangle \mid \langle\text{cmd}\rangle\|\langle\text{cmd}\rangle \mid \langle\text{event}\rangle \\ \langle\text{event}\rangle & ::= & \text{ID}\uparrow\ \mid \text{ID}\downarrow\ \mid \textbf{skip} \end{array}$$

Figure 2: Basic commands and their composition.

2.1.3 Guarded Commands

In addition to sequential and parallel composition, commands within a process can also be composed in *conflict* to specify a choice of behavior made by the environment. Conflict, or choice, is represented with a set of *guarded commands* (denoted $[\ G_1 \rightarrow C_1 \mid \ \ldots \mid G_n \rightarrow C_n\ ]$). The guard G_i of a guarded command is a boolean expression over a set of events. The events in this expression can be composed *conjunctively* (denoted $e_1 \wedge \ldots \wedge e_n$) in which the expression evaluates to true when the process has seen all the events in the set. Mutually exclusive events can also be composed *disjunctively* (denoted $e_1 \vee \ldots \vee e_n$) in which the expression evaluates to true when the process has seen exactly one event in the set. The expression may also include a combination of conjunctive and disjunctive clauses. Finally,

an expression may simply be the **skip** event which evaluates to true immediately. The expressions in our language differ from those used by Martin [44] in that ours are based on predicates on events rather than on predicates on signal values. This change in semantics is made because the representation used by our timing analysis algorithm is event-based.

When a guarded command is encountered, execution stalls until one of the guards G_i evaluates to true, after which the commands C_i associated with the guard that is satisfied are executed. If multiple guards evaluate to true, then one guard is nondeterministically chosen. Our synthesis procedure allows input choice but does not allow output choice, such as arbitration, so a specification must guarantee that either all expressions in a set of guarded commands are mutually exclusive, or that the first events in each set of non-mutually exclusive guarded commands is on a signal of type *input*. If an arbiter is needed in the design, it can be added as a special environment process.

A guarded command may also loop (denoted $G_i \rightarrow C_i; *$) [15]. If a guarded command that loops is selected, then after the set of commands is executed, control is returned to the beginning of the guarded command. This looping continues until a guarded command that does not loop is selected.

The timed HSE language makes use of abbreviations for two commonly used guarded command constructs [15]. The first is that a guarded command of the form $[G \rightarrow \textbf{skip}]$ may be written as $[G]$ which is called a *wait*. A wait simply specifies that the process must stall until the expression associated with the guard evaluates to true. The second abbreviation is that a guarded command of the form $[\textbf{skip} \rightarrow C; *]$ may be written as $*[C]$ which represents an infinite repetition of a set of commands. The parts of the grammar associated with guarded commands are shown in Figure 3.

$$
\begin{array}{rcl}
\langle\text{cmd}\rangle & ::= & [\langle\text{gdcmdset}\rangle] \mid [\langle\text{expr}\rangle] \mid\ *[\langle\text{cmd}\rangle] \\
\langle\text{gdcmdset}\rangle & ::= & \langle\text{gdcmdset}\rangle | \langle\text{gdcmdset}\rangle \mid \langle\text{gdcmd}\rangle \\
\langle\text{gdcmd}\rangle & ::= & \langle\text{expr}\rangle \rightarrow \langle\text{cmd}\rangle \mid \langle\text{expr}\rangle \rightarrow \langle\text{cmd}\rangle; * \\
\langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle \wedge \langle\text{expr}\rangle \mid \langle\text{expr}\rangle \vee \langle\text{expr}\rangle \mid \langle\text{event}\rangle
\end{array}
$$

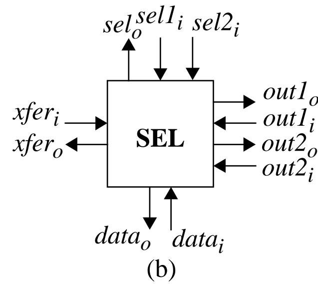
Figure 3: Guarded commands.

2.1.4 Example

As an example, consider the specification of a port selector (SEL) which is given in CSP in Figure 4(a). The CSP language includes all the constructs from the timed HSE language, as well as additional types of events to communicate on *ports*. A port is one side of a communication channel between two concurrent processes. A communication on a port is used for synchronization of the processes, and it may also be used to transmit data between them. Ports can be of either *passive* or *active* type. A port is passive if communications on the port are initiated by the environment process, and a port is active if communications are initiated by the process being designed. In the SEL, the *xfer* port is *passive* and all the other ports are *active*.

*[[$\overline{xfer}$ → (*data* ‖ *sel?(sel1, sel2)*);
[*sel1* → *out1; xfer*
| *sel2* → *out2; xfer*
]]

(a)



(b)

Figure 4: (a) CSP specification and (b) block diagram for a port selector (SEL).

The basic operation of the SEL is as follows. First, the SEL waits until it gets a request for a data transfer (i.e., $\overline{xfer}$), then it concurrently issues requests for the data to be transferred (i.e., *data*) and for the selection of an output port (i.e., *sel?(sel1,sel2)*). After the SEL receives the data and the port selection (i.e., *sel1* or *sel2*), it initiates the transfer of the data onto the selected output port (i.e., *out1* or *out2*) and then acknowledges the completion of the data transfer (i.e., *xfer*).

A timed HSE specification is derived from a CSP specification by translating all the communications on ports to their corresponding signal transitions that implement the communications. These communications can be implemented in many ways. The most common methods use either a *two-phase handshaking* or *four-phase handshaking* protocol. In both methods, a simple synchronization communication is implemented with two wires, one for requests and one for acknowledgments. In the two-phase method, both the rising and falling transitions represent requests and acknowledgments. It is called two-phase because a cycle involves two transitions, one request and one acknowledgment. In the four-phase method,

only one type of transition (either the rising or the falling transition) represents a request or acknowledgment, and before another request or acknowledgment the corresponding wire must return to its original value. Thus, this protocol requires four transitions in a cycle. While either protocol could be specified and implemented, we use the four-phase protocol because it typically results in simpler logic.

Returning to the SEL, the signal wires that implement the communications in the CSP specification are shown in the block diagram in Figure 4(b). For example, the *xfer* communication is implemented with two wires, one request wire $xfer_i$ and one acknowledge wire $xfer_o$. The port selection *sel* is implemented with three wires, one request wire sel_o and two data wires $sel1_i$ and $sel2_i$.

The first step in translating the CSP specification to a timed HSE specification is to create declarations for each signal wire that is needed to implement the communications in the CSP specification. Finding the delays, or *timing constraints*, to associate with the transitions on these signal wires is not a trivial task. The timing constraints for input signal transitions can usually be determined from interface specifications or datapath delay estimates. The timing constraints for output signal transitions, however, presents a "chicken and egg problem", since the timing constraints cannot be known until the circuit is synthesized, but the circuit cannot be synthesized without giving the timing constraints. The traditional delay-insensitive or speed-independent approaches assume no timing information. In other words, they assume that delays can be anywhere from 0 to infinity. This conservative assumption can often lead to unnecessarily complex circuit implementations. It is quite reasonable, however, to expect an automatic analysis of the given gate library to produce a safe estimate of the maximum delay for the gates in the library to be used, and by making some assumptions about the complexity of the synthesized logic, this can be used to set the upper bound of the timing constraint for each output signal transition. The lower bound of the timing constraint should usually be set to a very low value since optimizations could potentially reduce the gate to nothing more than a wire. After the circuit is generated, it must be back-annotated with timing information from the gate library and verified to be correct which is the subject of Chapter 7. If the circuit fails verification, it must be resynthesized with more conservative timing constraints (larger upper bounds and/or smaller lower bounds). In order to avoid resynthesis, conservative values should be used for timing constraints on output signal transitions. Of course, more aggressive estimates of the gate delays can be used leading potentially to better implementations, but

may require more iterations of the synthesis procedure. In the design of interface circuits and other controllers, inputs often are from off-chip or from a datapath. In these cases, the lower bound of the timing constraints on input signal transitions is large compared with the upper bound of the timing constraints on output signal transitions. Therefore, a conservative estimate for gate delays may not significantly affect the complexity of the timed circuit implementation.

The next step is to translate each communication on a port to its corresponding signal transitions. A communication on a passive port such as the *xfer* port is expanded as follows:

$$[xfer_i \uparrow]; xfer_o \uparrow; [xfer_i \downarrow]; xfer_o \downarrow$$

The placement of the first wait is dictated by the *probe* on the *xfer* port (i.e., $\overline{xfer}$). The probe is used to test if there is a pending communication on a passive port. The rest of the communication on the passive *xfer* port are expanded where the communication is completed (i.e., *xfer*).

A communication on an active port such as the *data* port is expanded as follows:

$$data_o \uparrow; [data_i \uparrow]; data_o \downarrow; [data_i \downarrow]$$

As an optimization, however, an active port is usually implemented using a *lazy-active* protocol [44] which is expanded as follows:

$$[data_i \downarrow]; data_o \uparrow; [data_i \uparrow]; data_o \downarrow$$

In this protocol, the reset of the four-phase handshake (i.e., the falling transition of the input wire) does not delay the execution until a new request on the active port is necessary. Note that since the $data_i$ wire is initially low, the first wait is *vacuous*. A vacuous event is one which is ignored in the initial cycle because it is the first event on that signal, and the event would set the value of the signal to its initial value. The two output ports *out1* and *out2* are similarly expanded. The *sel* port is slightly different in that there are two input wires associated with it that carry the data of which port is to be selected. Therefore, the request on this port must wait on either the falling transition of the $sel1_i$ wire or the $sel2_i$ wire depending on what port was used in the previous cycle. After a port selection is requested by rising the output wire sel_o, one of the two input wires $sel1_i$ or $sel2_i$ is set high by the environment. Part of the initial timed HSE specification for the SEL module is shown in Figure 5 including the signal declarations that implement the *sel* port, the control

process (*selctrl*) being designed, and the environment process (*sel*) which makes the choice of which output port to use.

module *SEL;*
input $sel1_i = \{\textbf{false}, \langle 40, 260; 2, 40\rangle\}$;
input $sel2_i = \{\textbf{false}, \langle 40, 260; 2, 40\rangle\}$;
output $sel_o = \{\textbf{false}, \langle 0, 20\rangle\}$;
etc.
process *selctrl;*
$*[\ [\ xfer_i \uparrow \rightarrow (([data_i \downarrow]; data_o \uparrow) \parallel ([sel1_i \downarrow \vee sel2_i \downarrow]; sel_o \uparrow)); [data_i \uparrow];$
$\quad [\ sel1_i \uparrow \rightarrow (sel_o \downarrow \parallel data_o \downarrow); [out1_i \downarrow]; out1_o \uparrow; [out1_i \uparrow]; out1_o \downarrow; xfer_o \uparrow; [xfer_i \downarrow]; xfer_o \downarrow$
$\quad |\ sel2_i \uparrow \rightarrow (sel_o \downarrow \parallel data_o \downarrow); [out2_i \downarrow]; out2_o \uparrow; [out2_i \uparrow]; out2_o \downarrow; xfer_o \uparrow; [xfer_i \downarrow]; xfer_o \downarrow$
$\ ]\ ]\ ]$
endprocess
process *sel;*
$*[\ [sel_o \uparrow]; [\ \textbf{skip} \rightarrow sel1_i \uparrow; [sel_o \downarrow]; sel1_i \downarrow$
$\qquad\qquad |\ \textbf{skip} \rightarrow sel2_i \uparrow; [sel_o \downarrow]; sel2_i \downarrow$
$\ ]\ ]$
endprocess
etc.
endmodule

Figure 5: Part of the timed HSE specification for the SEL.

The CSP specification dictates the ordering of communications on the ports, but many different timed HSE specifications using the corresponding signal wires could implement the communications. After expanding these communications, the resulting signal transitions can often be *reshuffled* to optimize the implementation [44]. In particular, there is typically a great deal of flexibility in the placement of the initiation of the reset of each four-phase handshake (i.e., the falling transition of the output signal wire for active ports). One possible reshuffling of these transitions is shown in Figure 6.

process *selctrl;*
$*[\ [\ xfer_i \uparrow \rightarrow (([data_i \downarrow]; data_o \uparrow) \parallel ([sel1_i \downarrow \vee sel2_i \downarrow]; sel_o \uparrow)); [data_i \uparrow];$
$\quad [\ sel1_i \uparrow \wedge out1_i \downarrow \rightarrow out1_o \uparrow; sel_o \downarrow; [out1_i \uparrow]; (xfer_o \uparrow \parallel data_o \downarrow); out1_o \downarrow; [xfer_i \downarrow]; xfer_o \downarrow$
$\quad |\ sel2_i \uparrow \wedge out2_i \downarrow \rightarrow out2_o \uparrow; sel_o \downarrow; [out2_i \uparrow]; (xfer_o \uparrow \parallel data_o \downarrow); out2_o \downarrow; [xfer_i \downarrow]; xfer_o \downarrow$
$\ ]\ ]\ ]$
endprocess

Figure 6: Reshuffling of the *selctrl* process.

2.2 Timed Event-Rule Structures

In order to define the behaviors specified by a module in the timed HSE specification language, we introduce *timed event-rule (ER) structures*, a variant of Winskel's event structures with timing. Event structures were introduced by Winskel [77], and timing has been added to them in several ways. Subrahmanyam added timing to event structures using temporal assertions [66]. Burns introduced timing in a deterministic version, the event-rule (ER) system, in which causality is represented using a set of rules, and a single delay value, rather than a bound, is associated with each rule [16]. In this section, we introduce timed ER structures which extend ER systems with bounded timing constraints and add conflict from event structures to model nondeterministic behavior (namely, environmental choice).

Timed ER structures are composed of a set of *atomic actions* (A), a set of *events* (E), a set of *rules* (R), and a *symmetric conflict relation* ($\#$). In timed circuits, the set of atomic actions A is the set of all possible signal transitions. The occurrence of an action is an event, and it is denoted (a, i) where a is the action and i is an *occurrence index* for the action. The first instance of this action has $i = 0$, and i increments with each subsequent instance. We partition the event set E into a set of input events (I) and a set of output events (O).

The rule set R is used to represent a causal dependence between two events. Each rule of the form $\langle e, f, l, u \rangle$ is composed of an *enabling event* e, an *enabled event* f, and a *bounded timing constraint* $\langle l, u \rangle$. Informally, a rule states that the enabled event cannot occur until the enabling event has occurred. Ignoring conflict for the moment, if two rules enable the same event then that event cannot occur until *both* enabling events have occurred. This causality requirement is termed *conjunctive*. The bounded timing constraint places a lower and upper bound on the timing of a rule. A rule is said to be *satisfied* if the amount of time which has passed since the enabling event has exceeded the lower bound of the rule. A rule is said to be *expired* if the amount of time which has passed since the enabling event has exceeded the upper bound of the rule. Again ignoring conflict, an event cannot occur until *all* rules enabling it are satisfied. An event must always occur before *every* rule enabling it has expired. Since an event may be enabled by multiple rules, it is possible that the difference in time between the enabled event and some enabling events exceed the upper bound of their timing constraints, but not for all enabling events. These timing constraints are the same as the *max constraints* described in [46] and the *type 2 arcs* described in [73].

The conflict relation is added 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. This models a form of disjunctive causality. *Inherently disjunctive* behavior, or true OR causality, cannot currently modeled, but we are investigating extending work by Lee in [40] to address this. 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.

The formal definition of our timed ER structure is given below in which $E = I \cup O$ and $\mathcal{N} = \{1, 2, 3, \ldots\}$:

Definition 2.2.1 *(Timed ER Structure) A timed ER structure is* $S = \langle A, I, O, R, \# \rangle$ *where*

1. *A is the set of atomic actions;*
2. *$I \subseteq A \times \mathcal{N}$ is the set of input events;*
3. *$O \subseteq A \times \mathcal{N}$ is the set of output events;*
4. *$R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\})$ is the set of rules;*
5. *$\# \subseteq E \times E$ is the conflict relation.*

Events are labeled using the function $L : E \rightarrow A$.

2.3 Timed Configurations

For a timed ER structure, we define the allowed behaviors specified by the structure using *timed configurations*. Winskel defined the allowed behaviors of event structures as subsets of events, or *configurations* [77]. In order to add timing, we introduce timed configurations in which each event is now paired with the time of its occurrence.

The first requirement for a subset of events to be a configuration is that it must be *conflict-free*. In other words, if two events are in conflict, it is not allowed for both of them to occur in a configuration. Winskel defined Con to be the set of finite conflict-free subsets of E, i.e. $Con \subseteq 2^E$, defined as follows:

$$Con = \{X \mid (X \subseteq E) \wedge (\forall e, e' \in X \,.\, \neg(e\#e'))\}.$$

In order to add timing, we define *TCon* to be the set of conflict-free subsets of events in which each event is paired with the real-valued time that the event occurred (i.e., $TCon \subseteq 2^{E \times \Re}$). To obtain the *Con* set from *TCon*, we define the function $untime : TCon \rightarrow Con$ in the obvious way.

The second requirement is that all events in the subset must be *time-secured*. Informally, this means that for each event in the set, all the events needed to enable the event are also in the set. To define this formally, we must first define when an event is enabled. The *untimed enabling relation* ($\vdash \subseteq Con \times E$) is defined as follows:

$$X \vdash f \quad \Leftrightarrow \quad [(\langle e, f, l, u \rangle \in R) \Rightarrow ((e \in X) \vee (\exists e' \in X \, . \, (e \# e') \wedge \langle e', f, l', u' \rangle \in R))].$$

Intuitively, this says given that the events in the set X have occurred that the event f is untimed-enabled. This is true when a set of non-conflicting enabling events in rules in which f is the enabled event are in the set X. To incorporate timing, we now define the *timed enabling relation* ($\vdash_t \subseteq TCon \times \Re \times E$) as follows:

$$Z \vdash_t f \quad \Leftrightarrow \quad [(untime(Z) \vdash f) \wedge (\forall (e, t') \in Z \, . \, \langle e, f, l, u \rangle \in R \Rightarrow t \geq t' + l)].$$

Intuitively, this says that given that the set of event-time pairs in Z have occurred and time has advanced to time t, the event f is timed-enabled. This is true when f is untimed-enabled, and at time t the lower bounds of all timing constraints have been satisfied. With this relation, we can now define $time\text{-}secured \subseteq TCon \times E$ as follows:

$$\begin{aligned} time\text{-}secured(Z, e) \quad \Leftrightarrow \quad & [\exists (e_0, t_0), \ldots, (e_n, t_n) \in Z \, . \, e_n = e \wedge \\ & \forall i \leq n \, . \, \{(e_0, t_0), \ldots, (e_{i-1}, t_{i-1})\} \vdash_{t_i} e_i]. \end{aligned}$$

The third requirement for a subset of events to be a configuration is that it is *non-expired*. This means that all events must occur before they are *expired*. An event is expired when for all the rules enabling it, the time since the enabling event has exceeded the upper bound of the timing constraint. We define a relation $expired \subseteq TCon \times E \times \Re$ as follows:

$$expired(Z, f, t) \quad \Leftrightarrow \quad [(Z \vdash_t f) \wedge (\forall (e', t') \in Z \, . \, \langle e', f, l, u \rangle \in R \Rightarrow t > t' + u)].$$

Using this relation, we say a timed configuration Z is non-expired if for all events either the event has occurred and was not expired when it occurred, or it has not occurred and is not expired at the latest time of any event occurrence in the configuration. We define the

relation *non-expired* $\subseteq TCon \times E$ as follows:

$$\textit{non-expired}(Z,f) \;\Leftrightarrow\; [(\exists t \,.\, (f,t) \in Z \wedge \neg \textit{expired}(Z,f,t)) \vee \neg \textit{expired}(Z,f,\max_{(e,t)\in Z}\{t\})].$$

Now, we can define all the timed configurations specified by a timed ER structure.

Definition 2.3.1 *(Timed configurations) For a timed ER structure $S = \langle A, I, O, R, \# \rangle$, a timed configuration of S is a subset of event-time pairs $Z \subseteq E \times \Re$ which is:*

1. *conflict-free: $Z \in TCon$,*
2. *time-secured: $\forall e \in \textit{untime}(Z) \,.\, \textit{time-secured}(Z,e)$, and*
3. *non-expired: $\forall f \in E \,.\, \textit{non-expired}(Z,f)$.*

The set of all configurations is $\mathcal{C}(S)$.

2.4 Interpreting the Specification Language

In order to interpret the behavior of a module described in the timed HSE specification language, we translate it to a timed ER structure. The procedure that we use is similar to the one proposed by Subrahmanyam [66]. The first step uses the declarations to initialize some functions to return the attributes declared for each signal. Next, each process is iteratively decomposed until it is made up of only events and *simple waits* that are composed on the operators specifying sequencing, concurrency, and choice. A simple wait is one which is composed of an expression that includes only a single event. In order to translate the decomposed specification to a timed ER structure, we need a function to compose two timed ER structures on each of the operations, and a function to rename a timed ER structure to resolve event name clashes before composition.

This section first describes a method to interpret *non-repetitive processes* which is then extended to interpret *repetitive processes*. The interpretation procedures given in this section are quite detailed for completeness, and they may be skimmed or skipped on first reading without loss of continuity.

2.4.1 Declarations

The declarations are used to assign attributes to actions. These attributes are accessible through functions of the form $f : A \rightarrow \textit{attr}$. For each declaration of the form:

$$\textit{type } s \;=\; \{\textit{initial}, \langle l_r, u_r; l_f, u_f \rangle\};,$$

we make the following assignments to initialize the functions *type*, *init*, and *delay*:

$$\begin{aligned} type(s\uparrow) = type(s\downarrow) &= type \\ init(s\uparrow) = init(s\downarrow) &= initial \\ delay(s\uparrow) &= \langle l_r, u_r \rangle \\ delay(s\downarrow) &= \langle l_f, u_f \rangle. \end{aligned}$$

2.4.2 Composition of Timed Event-Rule Structures

Each process is made up of a set of events and waits that are composed on operators specifying sequencing (;), concurrency ($\|$), and choice ($|$). Therefore, we need to define a means of composing two timed ER structures. To facilitate this composition, two subsets of the event set are added temporarily to the timed ER structure: *first* and *last*. Intuitively, the *first* set indicates which events are the first to occur in a timed ER structure, and the *last* set indicates which events are the last to occur. The composition of two timed ER structures $S_0 = \langle A_0, I_0, O_0, R_0, \#_0, first_0, last_0 \rangle$ and $S_1 = \langle A_1, I_1, O_1, R_1, \#_1, first_1, last_1 \rangle$ (i.e., $S_0\ op\ S_1$ where $op \in \{;, \|, |\}$) is defined as follows:

$$\begin{aligned} A &= A_0 \cup A_1 \\ I &= I_0 \cup I_1 - (O_0 \cup O_1) \\ O &= O_0 \cup O_1 \\ R &= R_0 \cup R_1 \cup \{\langle e, f, delay(L(f))\rangle \mid e \in last_0 \wedge f \in first_1 \wedge op = ;\} \\ \# &= \#_0 \cup \#_1 \cup \{(e, e') \mid (e \in O_0 \wedge e' \in O_1 \wedge op = \ | \ )\} \\ first &= \textbf{if } (first_0 = \emptyset \vee op = \| \vee op = |) \textbf{ then } first_0 \cup first_1 \textbf{ else } first_0 \\ last &= \textbf{if } (first_1 = \emptyset \vee last_1 = \emptyset \vee op = \| \vee op = |) \textbf{ then } last_0 \cup last_1 \textbf{ else } last_1 \end{aligned}$$

The sets of actions and output events are simply merged. The set of input events are also merged, but any events which are also output events are removed to keep the sets of input and output events disjoint. If there are no input events (i.e., $I = \emptyset$), we say the structure is *closed*. In order to perform synthesis, we require the structure obtained from the complete specification to be closed. The rule sets are similarly combined, but in the case in which $op = ;$ new rules are added from the last events in S_0 (i.e., the events in the set $last_0$) to the first events in S_1 (i.e., the events in the set $first_1$). The conflict sets are also merged, and if $op = |$ then every output event in S_0 is set to conflict with every

output event in S_1. Finally, new *first* and *last* sets are created. If the structures are being composed in parallel or in conflict, the sets are created by simply taking the union of the sets from each structure. If the structures are being composed in sequence, then in most cases the *first* set equals $first_0$, and the *last* set equals $last_1$. The exception is if $first_0$ is empty then the *first* set equals $first_1$, and if either $first_1$ or $last_1$ is empty than the *last* set is the union of the two last sets from the two structures.

2.4.3 Renaming of Timed Event-Rule Structures

When composing structures sequentially or in conflict, multiple occurrences of events with the same name are not allowed. Therefore, before doing the composition, we first resolve any name clashes using the function *rename* which takes two structures and returns the second structure with event names changed such that they do not clash with event names in the first structure. The function $rename(S_0, S_1)$ is defined as follows:

$$
\begin{aligned}
A &= A_1 \\
I &= \{rename(E_0, e) \mid e \in I_1\} \\
O &= \{rename(E_0, e) \mid e \in O_1\} \\
R &= \{\langle rename(E_0, e), rename(E_0, f), l, u\rangle \mid \langle e, f, l, u\rangle \in R_1\} \\
\# &= \{(rename(E_0, e), rename(E_0, e')) \mid e\#e'\} \\
first &= \{rename(E_0, e) \mid e \in first_1\} \\
last &= \{rename(E_0, e) \mid e \in last_1\}
\end{aligned}
$$

The function *rename* is overloaded above to take a set of events E and a single event (a, i), and it renames (a, i) if there is a name clash with an event in the set E as follows:

$$
\begin{aligned}
rename(E, (a, i)) = \; & \textbf{if } (\forall k (a, k) \notin E) \textbf{ then } (a, i) \textbf{ else } (a, i + j) \\
& \quad \textbf{where } (a, j - 1) \in E \wedge (a, j) \notin E.
\end{aligned}
$$

2.4.4 Interpretation of a Non-Repetitive Process

A non-repetitive process is one which does not contain any looping constructs (i.e., guarded commands of the form $G \rightarrow C; *$ or the infinite loop construct $*[C]$). These constructs are addressed in the next subsection. To interpret a non-repetitive process, we define the function *TERS* which takes a timed HSE specification and returns a timed ER structure

of the form: $S = \langle A, I, O, R, \#, \mathit{first}, \mathit{last} \rangle$. This function iteratively decomposes the timed HSE specification into events and simple waits that are composed on the operators, and it is defined as follows:

$$\begin{aligned}
TERS(p;q) &= TERS(p); \mathit{rename}(TERS(p), TERS(q)) \\
TERS(p\|q) &= TERS(p)\|TERS(q) \\
TERS([p \mid q]) &= TERS([p]) \mid \mathit{rename}(TERS([p]), TERS([q])) \\
TERS([p \rightarrow q]) &= TERS([p]); \mathit{rename}(TERS([p]), TERS(q)) \\
TERS([p \vee q]) &= TERS([p]) \mid \mathit{rename}(TERS([p]), TERS([q])) \\
TERS([p \wedge q]) &= TERS([p])\|TERS([q]) \\
TERS([a]) &= \langle \{a\}, \{(a,1)\}, \emptyset, \emptyset, \emptyset, \emptyset, \{(a,1)\} \rangle \\
TERS(a) &= \langle \{a\}, \emptyset, \{(a,1)\}, \emptyset, \emptyset, \{(a,1)\}, \{(a,1)\} \rangle \\
TERS([\mathbf{skip}]) &= \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \\
TERS(\mathbf{skip}) &= \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle
\end{aligned}$$

where p and q are segments of a process, and a is an action.

The first rule simply states that the structure for two sets of commands p and q composed sequentially is obtained by finding the structure for p and q, renaming the events in the structure for q, if necessary, and composing these structures using the sequencing operation (;). When composing p and q in parallel, a structure is again first found for each, but they are composed using the parallel operation ($\|$) and renaming is not done. In order to generate the structure for a pair of guarded commands p and q, the structure for each guarded command $[p]$ and $[q]$ is found individually, the events in the structure for $[q]$ are renamed, if necessary, and the resulting structures are composed using the conflict operation ($|$). For an individual guarded command ($[p \rightarrow q]$), a structure is obtained for a wait $[p]$, and it is composed sequentially with a renamed version of the structure for q. The next two rules are for evaluating expressions in waits. The first says that a disjunct in a wait is defined to be semantically equivalent to two waits in conflict. The second says a conjunct in a wait is defined to be semantically equivalent to two waits in parallel. The last four translate events and simple waits into timed ER structures. First, if the input to the function is a simple wait on any event other than **skip**, the function returns a structure with a single action, a single input event, and the *last* set initialized to include the input event. Next, if the input to the function is an event other than **skip**, the function returns a structure with

a single action, a single output event, and both the *first* and *last* sets initialized to include the output event. Finally, if the input to the function is a simple wait on **skip** or the **skip** event, the function returns an empty structure.

2.4.5 Interpretation of a Repetitive Process

If a process is repetitive, then the timed ER structure describing its behavior is infinite. Due to its repetitive nature, however, this infinite behavior can be described with a finite model by adding an additional set of rules R' and an additional set of conflicts $\#'$. A *loop* set is also added temporarily to keep track of the last events before control loops back. When a timed ER structure is created, these sets are all initialized to the empty set. The *rename* function is modified in the obvious way to accommodate these new sets. To generate these sets, the composition operator is modified as follows:

$$\begin{aligned} R' &= R'_0 \cup R'_1 \cup \{\langle e, f, delay(L(f))\rangle \mid e \in loop_1 \wedge f \in first_1 \wedge op = \,;\} \\ \#' &= \#'_0 \cup \#'_1 \cup \{(e, e') \mid (e \in loop_1 \wedge e' \in last_0 \wedge op = \,;)\} \\ loop &= \textbf{if } (op = \| \vee op = |) \textbf{ then } loop_0 \cup loop_1 \textbf{ else } \emptyset. \end{aligned}$$

The R' set is found by first taking the union of the corresponding sets from the structures that are being composed, and then when $op = ;$, new rules are added from events in the $loop_1$ set to the $first_1$ set which creates a loop in the structure. Also, if $op = ;$ then the events in $last_0$ are set to conflict with the events in $loop_1$. As for the *loop* set, the events in the loop sets from the structures being composed in parallel or in conflict are simply merged and initialized to the empty set when composed in sequence. Finally, there is one more special case of composition in which S_0 is being composed in sequence with '$*$' (or equivalently, in the case of the '$*[\ ]$' construct). In this case, the structure S_0 is returned with the *loop* set equal to $last_0$ and the *last* set equal to the empty set.

With a timed ER structure of the form $S_0 = \langle A_0, I_0, O_0, R_0, \#_0, R'_0, \#'_0\rangle$, we can inductively define the infinite behavior specified by a repetitive process as follows:

$$S_i = loop(S_0, S_0 \| rename(S_0, S_{i-1}))$$

where $loop(S_0, S_1)$ is defined as follows:

$$\begin{aligned} R &= R_1 \cup \{\langle e, rename(E_0, f), l, u\rangle \mid \langle e, f, l, u\rangle \in R'_0\} \\ \# &= \#_1 \cup \{(e, rename(E_0, e')) \mid e \#'_0 e'\}. \end{aligned}$$

2.4.6 Vacuous Events

In the previous sections, we ignored the possibility that some events may be vacuous in the first cycle, such as the first wait in the lazy-active protocol described earlier. We utilize the additional sets described in the previous section to model vacuous events. In order to detect that an event is vacuous, a bitvector v is used to represent the possibility of each signal having a vacuous event on it. Before beginning the interpretation of each process, all elements in v are initialized to *true*, and as each action appears non-vacuously, the appropriate element is set to false. To determine if an action is vacuous, we define the following function:

$$\begin{aligned} \mathit{vacuous}(v,a) &= \textbf{if } v(a) \textbf{ and } ((\mathit{init}(a) \textbf{ and } a = s+) \textbf{ or } (\neg \mathit{init}(a) \textbf{ and } a = s-)) \\ &\quad \textbf{then } \mathit{true} \textbf{ else } \mathit{false}. \end{aligned}$$

This information is used to modify the composition function as follows:

$$\begin{aligned} R &= R_0 \cup R_1 \cup \{\langle e, f, \mathit{delay}(L(f))\rangle \mid e \in \mathit{last}_0 \wedge f \in \mathit{first}_1 \wedge \mathit{op} = \,;\, \wedge \neg \mathit{vacuous}(v,e)\} \\ R' &= R'_0 \cup R'_1 \cup \{\langle e, f, \mathit{delay}(L(f))\rangle \mid e \in \mathit{loop}_1 \wedge f \in \mathit{first}_1 \wedge \mathit{op} = \,;\,\} \\ &\quad \cup \{\langle e, f, \mathit{delay}(L(f))\rangle \mid e \in \mathit{last}_0 \wedge f \in \mathit{first}_1 \wedge \mathit{op} = \,;\, \wedge \mathit{vacuous}(v,e)\} \end{aligned}$$

This puts all rules with vacuous enabling events into the R' set, so they are enabling events in the next cycle.

2.4.7 Interpretation of a Module

In order to obtain the timed ER structure for a complete module, it is now simply a matter of composing all the individual processes in parallel, i.e.,

$$\mathit{TERS}(P\ Q) = \mathit{TERS}(P) \| \mathit{TERS}(Q)$$

where P and Q are processes.

2.4.8 Example

The structure $S_0 = \langle A_0, I_0, O_0, R_0, \#_0, R'_0, \#'_0 \rangle$ that is obtained for the environmental selection process *sel* from the SEL is shown below:

$$
\begin{aligned}
A_0 &= \{sel_o\uparrow, sel_o\downarrow, sel1_i\uparrow, sel1_i\downarrow, sel2_i\uparrow, sel2_i\downarrow\} \\
I_0 &= \{(sel_o\uparrow,1),(sel_o\downarrow,1),(sel_o\downarrow,2)\} \\
O_0 &= \{(sel1_i\uparrow,1),(sel1_i\downarrow,1),(sel2_i\uparrow,1),(sel2_i\downarrow,1)\} \\
R_0 &= \{\langle(sel_o\uparrow,1),(sel1_i\uparrow,1),40,260\rangle,\langle(sel_o\uparrow,1),(sel2_i\uparrow,1),40,260\rangle, \\
&\quad \langle(sel1_i\uparrow,1),(sel1_i\downarrow,1),2,40\rangle,\langle(sel2_i\uparrow,1),(sel2_i\downarrow,1),2,40\rangle, \\
&\quad \langle(sel_o\downarrow,1),(sel1_i\downarrow,1),2,40\rangle,\langle(sel_o\downarrow,2),(sel2_i\downarrow,1),2,40\rangle\} \\
\#_0 &= \{\langle(sel1_i\uparrow,1),(sel2_i\uparrow,1)\rangle,\langle(sel1_i\uparrow,1),(sel2_i\downarrow,1)\rangle, \\
&\quad \langle(sel1_i\downarrow,1),(sel2_i\uparrow,1)\rangle,\langle(sel1_i\downarrow,1),(sel2_i\downarrow,1)\rangle\} \\
R'_0 &= R'_0 + \{\langle(sel1_i\downarrow,1),(sel1_i\uparrow,1),40,260\rangle,\langle(sel1_i\downarrow,1),(sel2_i\uparrow,1),40,260\rangle, \\
&\qquad \langle(sel2_i\downarrow,1),(sel1_i\uparrow,1),40,260\rangle,\langle(sel2_i\downarrow,1),(sel2_i\uparrow,1),40,260\rangle\} \\
\#'_0 &= \emptyset.
\end{aligned}
$$

Appendix

The formal syntax rules for our timed HSE language are given using BNF notation in Figure 7. As with the abstract grammar, the left-hand and right-hand side of a syntax rule is separated using '::=', and alternatives are separated with '|'. There are, however, some additional constructs such as the bracket pair [...] which means optional, the brace pair { ...} which means repeat zero or more times, and the selection construct '(...|...)' which is used to indicate a choice of options. Again, keywords are boldface and syntax rules are enclosed in angle brackets '⟨...⟩'. ID represents an identifier which is a string of alpha-numeric characters starting with a letter. INT represents an integer or the keyword **inf** or **infinity**. Finally, the symbols used in the language are enclosed in single quotes. In Figure 8, the SEL is given as it would appear as input to the CAD tool `ATACS`.

There are two additional constructs that are added here which are used to facilitate the specification of timing parameters. The first is delay macros which can be defined and then used in signal declarations later. The second construct is a delay override which is specified using a delay before an event to override the default delay given in the declaration for this particular occurrence of the event.

Within `ATACS`, the command *compile* ⟨*filename*⟩ compiles a timed HSE specification given in the file named ⟨*filename*⟩*.hse* into a timed ER structure which is stored into a file named ⟨*filename*⟩*.er*. `ATACS` can also accept a timed ER structure as input directly which is loaded with the command *loader* ⟨*filename*⟩. The format of the file is shown in Figure 9. First, comments begin with the '#' character which causes the program to ignore the rest of the line. The header of the file gives the numbers of each type of entry that follows. This includes the total number of events, the number which are input events, the number of rules, and the number of conflicts. The next entry '.s' is used to specify the initial state which is given as a bitvector of 0's and 1's with a length equal to the number of signals in the specification. The first event is always the *reset* event, the next set of events are those on inputs, and the last set of events are the ones on outputs. Each event other than *reset* is composed of an action and an occurrence number separated by a '/'. Each rule is composed of an enabling event, enabled event, indication of which rule set it is in ('1' if it is in R', '0' if it is in R), a lower bound, and an upper bound of the timing constraint on the rule. Finally, each conflict is a pair of events. The timed ER structure for the *sel* process from the SEL is shown in Figure 10.

⟨module⟩ ::= **module** ID ‘;’{⟨decls⟩}{⟨processes⟩} **endmodule**

⟨decls⟩ ::= ⟨decl⟩{⟨decl⟩}

⟨decl⟩ ::= ⟨delaydecl⟩ | ⟨sigdecl⟩

⟨delaydecl⟩ ::= **delay** ID ‘ = ’⟨delay⟩‘; ’

⟨sigdecl⟩ ::= (**input** | **output**) ID [= ‘{’⟨sigspec⟩‘}’]‘; ’

⟨sigspec⟩ ::= (**true** | **false**) | ⟨delay⟩ | (**true** | **false**)‘, ’⟨delay⟩

⟨delay⟩ ::= ‘ < ’ INT, INT [‘;’ INT, INT] ‘ > ’ | ID

⟨processes⟩ ::= ⟨process⟩{⟨process⟩}

⟨process⟩ ::= **process** ID ‘;’{⟨cmds⟩} **endprocess**

⟨cmds⟩ ::= ⟨cmd⟩{‘; ’⟨cmd⟩}

⟨cmd⟩ ::= [⟨delay⟩]⟨event⟩ | ⟨parastruct⟩ | ⟨gdcmdstruct⟩

⟨event⟩ ::= ID + | ID − | **skip**

⟨parastruct⟩ ::= ‘(’⟨parallel⟩‘)’

⟨parallel⟩ ::= ⟨cmds⟩{‘‖’⟨cmds⟩}

⟨gdcmdstruct⟩ ::= ‘[’⟨gdcmdset⟩‘]’ | ‘ ∗ ’‘[’⟨cmds⟩‘]’ | ‘[’⟨expr⟩‘]’

⟨gdcmdset⟩ ::= ⟨gdcmd⟩{‘|’⟨gdcmd⟩}

⟨gdcmd⟩ ::= ⟨expr⟩‘− > ’⟨cmds⟩[‘; ’‘ ∗ ’]

⟨expr⟩ ::= ⟨conjunct⟩{‘|’⟨conjunct⟩}

⟨conjunct⟩ ::= ⟨literal⟩{‘&’⟨literal⟩}

⟨literal⟩ ::= ⟨event⟩ | ‘(’⟨expr⟩‘)’

Figure 7: Complete BNF description for the timed HSE specification language.

```
module SEL;
delay  gatedelay =< 0, 20 >;
delay  seldelay =< 40, 260; 2, 40 >;
etc.
input   sel1i = {false, seldelay};
input   sel2i = {false, seldelay};
output  selo = {false, gatedelay};
etc.
process selctrl;
*[ [ xferi+  -> (([datai-]; datao+) || ([sel1i-  | sel2i-]; selo+)); [datai+];
   [  sel1i+  & out1i-  -> out1o+; selo-; [out1i+]; (xfero+  || datao-); out1o-;
                             [xferi-]; xfero-
   |  sel2i+  & out2i-  -> out2o + selo-; [out2i+]; (xfero+  || datao-); out2o-;
                             [xferi-]; xfero-
 ] ] ]
endprocess
process sel;
*[ [selo+]; [ skip -> sel1i+; [selo-]; sel1i-
            | skip -> sel2i+; [selo-]; sel2i-
 ] ]
endprocess
etc.
endmodule
```

Figure 8: Part of the timed HSE specification for the SEL.

```
.e INT  # Number of events
.i INT  # Number of input events
.r INT  # Number of rules
.c INT  # Number of conflicts
.s  ⟨initial state⟩
reset
# List of input events
{ID'+'|'-')'/'INT}
# List of output events
{ID('+'|'-')'/'INT}
# List of rules
ID('+'|'-')'/'INT ID('+'|'-')'/'INT INT INT INT
# List of conflicts
ID('+'|'-')'/'INT ID('+'|'-')'/'INT
```

Figure 9: Format for a timed ER structure.

```
.e 9
.i 4
.r 10
.c 4
.s 000
reset
# List of input events
sel1i+/1 sel1i-/1
sel2i+/1 sel2i-/1
# List of output events
selo+/1 selo-/1
selo+/2 selo-/2
# List of rules in R
selo+/1 sel1i+/1 0 40 260
selo+/1 sel2i+/1 0 40 260
sel1i+/1 sel1i-/1 0 2 40
sel2i+/1 sel2i-/1 0 2 40
selo-/1 sel1i-/1 0 2 40
selo-/2 sel2i-/1 0 2 40
# List of rules in R'
sel1i-/1 sel1i+/1 1 40 260
sel1i-/1 sel2i+/1 1 40 260
sel2i-/1 sel1i+/1 1 40 260
sel2i-/1 sel2i+/1 1 40 260
# List of conflicts
sel1i+/1 sel2i+/1
sel1i+/1 sel2i-/1
sel1i-/1 sel2i+/1
sel1i-/1 sel2i-/1
```

Figure 10: Timed ER structure for the *sel* process from the SEL.

Chapter 3

Timing Analysis

In a home it is the site that matters;	居善地
in quality of mind it is depth that matters;	心善淵
in an ally it is benevolence that matters;	與善仁
in speech it is good faith that matters;	言善信
in government it is order that matters;	正善治
in affairs it is ability that matters;	事善能
in action it is timeliness that matters.	動善時
—Lao Zi	一老子

The basic idea behind synthesis and verification methods that use explicit state space exploration is that, if the reachable state space is finite or has a finite representation, only a finite subset of the possible behaviors needs to be considered to compute the complete set of reachable states. In our timed specifications, the occurrence times associated with events can take on real values, so there are an infinite number of timed states in the system. In order to perform explicit state space exploration, it is necessary to use a timing analysis algorithm to construct a finite representation of this infinite state space.

In this chapter, we describe two timing analysis algorithms that we developed and applied to timed state space exploration. The first technique is an efficient, heuristic algorithm which determines the minimum and maximum time difference between any two events in a *conflict-free* timed ER structure and uses this information to guide state space exploration. Since only conflict-free timed ER structures can be analyzed, this approach is limited to deterministic specifications. Therefore, we also introduce another technique, *partial order timing*, which makes use of *geometric region* representations of the timed state space and

partial order information to guide their creation. Using this procedure, any timed ER structure can be analyzed, but in order to do so, it must first be transformed into an *orbital net* representation that satisfies certain properties. This transformation procedure is also described.

3.1 Constraint graphs

As described in the previous chapter, an infinite timed ER structure can be specified using a finite representation of the form $S_0 = \langle A_0, I_0, O_0, R_0, \#_0, R'_0, \#'_0 \rangle$. If this structure is conflict-free (i.e., the conflict sets $\#_0$ and $\#'_0$ are empty), it can be fully described with a *cyclic constraint graph* which is a weighted marked graph in which the vertices are the events, the arcs are the rules, and the weights are the bounded timing constraints. If a rule is in R'_0, it is *initially marked* which means that a token is placed on the arc corresponding to the rule to indicate that the rule is initially untimed enabled. Each rule of the form $\langle e, f, l, u \rangle$ is represented in the graph with an arc connecting the enabling event e to the enabled event f. The arc is weighted with the bounded timing constraint $\langle l, u \rangle$. In other words, each rule corresponds to a graph segment, $e \xrightarrow{\langle l,u \rangle} f$ ($e \overset{\langle l,u \rangle}{\bullet\!\!\rightarrow} f$, if the rule is in R'_0). A cyclic constraint graph is essentially a signal transition graph (STG) [17] in which timing constraints have been added to the arcs.

As an example, a SCSI protocol controller, originally specified with a STG [18], is specified by its timed HSE specification in Figure 11. The timed ER structure for this specification is shown as a cyclic constraint graph in Figure 12. Note that as an optimization a simple analysis of the graph determines that some rules can be removed without changing the specified behavior. In particular, the following redundant rules from R_0 are not depicted in Figure 12:

$$\langle go\uparrow, go\downarrow, 20, 50 \rangle$$
$$\langle ack\downarrow, ack\uparrow, 20, 50 \rangle,$$

and the following redundant rules from R'_0 are also not depicted:

$$\langle go\downarrow, go\uparrow, 20, 50 \rangle$$
$$\langle ack\uparrow, ack\downarrow, 20, 50 \rangle$$

In general, a rule $\langle e, f, l, u \rangle$ in R_0 can be removed if there exists an alternative path from e to f which traverses only arcs from rules in R_0, the total minimum delay along this path is

greater than or equal to l, and the total maximum delay along this path is greater than or equal to u. If the rule is from R'_0, then the alternative path must include one and only one arc from a rule in R'_0 and all other arcs from R_0.

module *scsi*;
input $ack = \{$ **true**,$\langle 20, 50\rangle \}$;
input $go = \{$ **false**,$\langle 20, 50\rangle \}$;
output $req = \{$ **true**,$\langle 0, 5\rangle \}$;
output $rdy = \{$ **false**,$\langle 0, 5\rangle \}$;
output $q = \{$ **true**,$\langle 0, 5\rangle \}$;
process *scsictrl*;
$*[\ req\downarrow; rdy\uparrow; q\downarrow; [go\uparrow]; rdy\downarrow; [ack\downarrow]; req\uparrow; [go\downarrow]; q\uparrow; [ack\uparrow]\ ]$
endprocess
process *ackenv*;
$*[\ [req\downarrow]; ack\downarrow; [req\uparrow]; ack\uparrow\ ]$
endprocess
process *goenv*;
$*[\ [rdy\uparrow]; go\uparrow; [rdy\downarrow]; go\downarrow\ ]$
endprocess
endmodule

Figure 11: Timed HSE specification for a SCSI protocol controller.

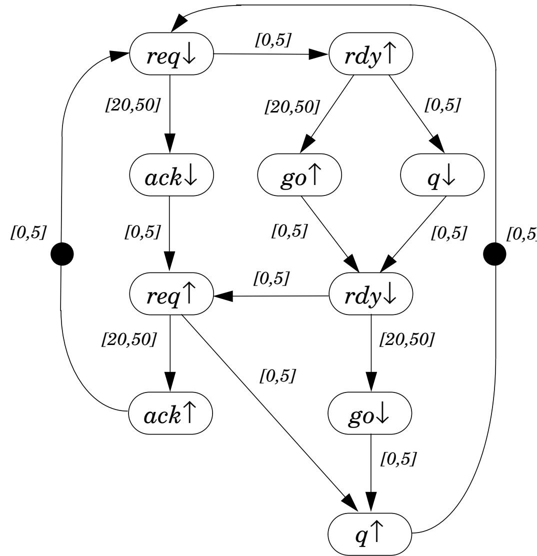


Figure 12: Cyclic constraint graph for a SCSI protocol controller.

A requirement for the timing analysis algorithm described in the next section is that the cyclic constraint graph is *well-formed*. A cyclic constraint graph is well-formed if it is strongly connected, every cycle has at least one arc which is initially marked, and for every event there exists a cycle including the event in which there is just one initially marked arc. Many specifications are not well-formed, but such specifications can often be analyzed by transforming them into ones which are well-formed.

An infinite conflict-free timed ER structure is represented with an *infinite acyclic constraint graph*. Each event e in the cyclic constraint graph can be mapped onto an infinite number of events in the acyclic constraint graph of the form $\langle e, i\rangle$ where i is used to denote each separate cycle from the unfolding of the cyclic constraint graph. The first cycle has $i = 0$, and i increments with each following occurrence. In the infinite acyclic constraint graph, each rule $\langle e, f, l, u\rangle$ corresponds to an infinite number of graph segments of the form: $\langle e, i\rangle \stackrel{\langle l,u\rangle}{\longrightarrow} \langle f, i\rangle$ (if the rule is in R_0' then it is of the form $\langle e, i-1\rangle \stackrel{\langle l,u\rangle}{\longrightarrow} \langle f, i\rangle$).

A special *reset* event is added to the set of events in order to model the reset of the circuit. For each initially marked rule (i.e., each rule in R_0') with enabled event f, a *reset rule* is added between the *reset* event and the event f. This rule models special timing constraints on the initial occurrence of the event f. The default timing constraint has a lower bound equal to the minimum of all initially marked rules with enabled event f and the upper bound is the maximum. Effectively, the acyclic constraint graph is constructed by cutting the cyclic constraint graph at the initial marking and unfolding the graph an infinite number of cycles. Part of the unfolded infinite acyclic constraint graph for the SCSI protocol controller is shown in Figure 13.

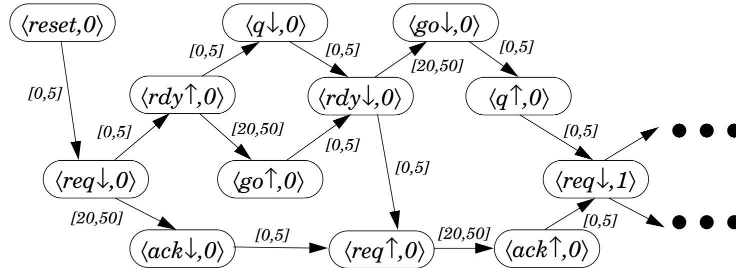


Figure 13: Part of the acyclic constraint graph for the SCSI protocol controller.

3.2 Estimating the Worst-Case Time Difference

In order to synthesize timed circuits, timing analysis must be used to deduce the timing information necessary to compute the reachable state space. For any particular state encountered while exploring the state space described by a cyclic constraint graph, there may be many possible next states depending on what is the next event that occurs. Timing analysis can be used to reduce the number of possible next states by showing that certain events which appear concurrent in the specification are actually ordered. The timing information which is required for this check is the minimum and maximum difference in time between any two events in the cyclic constraint graph. Polynomial-time algorithms have been developed [46] [73] to determine the difference in time between any two events in an acyclic graph. Circuit specifications, however, are normally cyclic. Therefore, to apply these algorithms to circuit synthesis, these results must be extended to handle cyclic specifications. Exponential-time algorithms have been proposed that find time differences in cyclic graphs [3, 35]. In this section, we propose a polynomial-time heuristic algorithm which is sufficient for the analysis of conflict-free timed ER structures. Our algorithm unfolds the cyclic graph into an infinite acyclic graph and then examines only two finite acyclic subgraphs of the infinite graph to determine a sufficient bound on the time difference between two events.

3.2.1 Worst-Case Time Difference

A *time difference* is a bound in the amount of time between two events in the cyclic constraint graph for a particular cycle. The *worst-case time difference* is a bound on the minimum and maximum difference in time between two events for any cycle.

Definition 3.2.1 *(Time Difference) Given two events and the cycles of their occurrence* $\langle u, i-j \rangle$ *and* $\langle v, i \rangle$ *where* $j \geq 0$ *is their cycle offset, the time difference between these two events is the strongest bound* $[L_i, U_i]$ *such that:*

$$L_i \leq t(\langle v, i \rangle) - t(\langle u, i-j \rangle) \leq U_i$$

Definition 3.2.2 *(Worst-Case Time Difference) Given two events* u *and* v *from a cyclic constraint graph and the cycle offset between them* j *where* $j \geq 0$*, the worst-case time difference between these two events in any cycle is* $[L, U]$ *defined to be:*

$$L = \min_{i \geq j}\{L_i\} \text{ and } U = \max_{i \geq j}\{U_i\},$$

where $[L_i, U_i]$ *is the time difference between* u *and* v *with offset* j *in each cycle* i*.*

3.2.2 Algorithm to Estimate the Worst-Case Time Difference

A pair of events from a cyclic constraint graph can appear in an infinite number of cycles in the corresponding acyclic constraint graph; however, it is possible to analyze a finite number of cycles to find a sufficient *estimate of the worst-case time difference.*

Definition 3.2.3 *(Estimate of the Worst-Case Time Difference) Given the worst-case time difference* $[L, U]$ *between two events from a cyclic constraint graph, an estimate of the worst-case time difference is any* $[L', U']$ *such that* $L' \leq L$ *and* $U' \geq U$*.*

Given two events u and v from a cyclic constraint graph and a cycle offset between them j, Algorithm 3.2.1 determines an estimate of the worst-case time difference between them by constructing two finite acyclic subgraphs to be analyzed by Algorithm 3.2.2. The first subgraph includes only events from cycles $i-1$ and i for some arbitrary value of $i > 0$. A *source* event is added to this subgraph, and for each rule in R_0', an additional arc is added from the *source* event to the enabled event with a timing constraint of $[0, \infty]$. This construction guarantees that no timing assumptions are made about previous cycles which are not modeled in our finite graph. For the special case when $i = 0$, another subgraph is constructed which includes only events from cycle 0. We prove later that the analysis of these two subgraphs yields an estimate of the worst-case time difference.

These two subgraphs are acyclic and finite so the algorithms described in [46] and [73] can be used to find the time difference between any two events $\langle u, i-j \rangle$ and $\langle v, i \rangle$ in these graphs. The function *MaxDiff* (defined recursively in Algorithm 3.2.3 [73]) is used to find the upper bound of the time difference U_i. *MaxDiff* is also used to find the minimum time difference L_i since *MinDiff*$(\langle u, i-j \rangle, \langle v, i \rangle) = (-1) *$ *MaxDiff*$(\langle v, i \rangle, \langle u, i-j \rangle)$ [46] [73]. The estimate of the worst-case time difference returned by Algorithm 3.2.1 is the minimum of the lower bounds and the maximum of the upper bounds of the time differences for the ith and 0th cycle. Since the worst-case time difference is only defined over values of i where $i \geq j$, the 0th occurrence only needs to be considered if $j = 0$. As an optimization, when this algorithm is called repeatedly the graphs are created only once for a given circuit, and once a time difference is calculated for a particular pair of events, it is stored in a table.

For the example shown in Figure 12, the estimate of the worst-case time difference found by Algorithm 3.2.1 between the two events $rdy \downarrow$ and $q \downarrow$ with cycle offset $j = 0$ is the bound $[15, 55]$. This means that $rdy \downarrow$ always occurs at least 15 units of time after $q \downarrow$, but no more than 55 units of time after $q \downarrow$.

Algorithm 3.2.1 (Estimate the worst-case time difference in a cyclic graph)
bound WCTimeDiff(timed ER structure $\langle A_0, E_0, R_0, R_0' \rangle$*; events* u, v*; cycle offset* j*)* {
  if $(j > 1)$ **then return**$([-\infty, \infty])$;
  else {
    construct subgraph G *from* $\langle A_0, E_0, R_0, R_0' \rangle$ *using only events with cycle indices*
      $i-1$ *and* i *for an arbitrary* $i > 0$ *and exclude rules with* reset *enabling event;*
    add source *event to graph* G*;*
    foreach *rule of the form* $\langle e, f, l, u \rangle$ *in* R_0'*, add an arc from* source *to* $\langle f, i-1 \rangle$
      weighted with $\langle 0, \infty \rangle$*;*
    $[L_i, U_i] = TimeDiff(G, \langle u, i-j \rangle, \langle v, i \rangle)$;
    if $(j == 1)$ **then return**$([L_i, U_i])$;
    else {
      construct subgraph G' *from* $\langle A_0, E_0, R_0, R_0' \rangle$ *using only events with cycle index* 0*;*
      $[L_0, U_0] = TimeDiff(G', \langle u, 0 \rangle, \langle v, 0 \rangle)$;
      $L' = min(L_i, L_0)$;
      $U' = max(U_i, U_0)$;
      return$([L', U'])$;
} } }

Figure 14: Algorithm to find an estimate of the worst-case time difference in a cyclic graph.

Algorithm 3.2.2 (Find a time difference in an acyclic graph)
bound TimeDiff(acyclic graph G*; events* $\langle u, i-j \rangle$, $\langle v, i \rangle$*)* {
  $L_i = (-1) * MaxDiff(G, \langle v, i \rangle, \langle u, i-j \rangle)$;
  $U_i = MaxDiff(G, \langle u, i-j \rangle, \langle v, i \rangle)$;
  return$([L_i, U_i])$;
}

Figure 15: Algorithm to find a time difference in an acyclic graph.

Algorithm 3.2.3 (Find a maximum time difference in an acyclic graph)
int MaxDiff(acyclic graph G; events $\langle u, i-j\rangle$, $\langle v, i\rangle$*)* {
  $maxdiff = \max_{\langle e,i-\varepsilon\rangle \xrightarrow{\langle l,u\rangle} \langle v,i\rangle \in G} \{MaxDiff(G, \langle u, i-j\rangle, \langle e, i-\varepsilon\rangle) + u\};$
  if *there is a path from* $\langle v, i\rangle$ *to* $\langle u, i-j\rangle$ **then**
    $maxdiff = \min\{ \min_{\langle e,i-j-\varepsilon\rangle \xrightarrow{\langle l,u\rangle} \langle u,i-j\rangle \in G} \{MaxDiff(G, \langle e, i-j-\varepsilon\rangle, \langle v, i\rangle) + l\}, maxdiff\};$
  return*(maxdiff);*
}

Figure 16: Algorithm to find a maximum time difference in an acyclic graph.

3.2.3 Proof of Correctness

Theorem 3.2.1 shows that the bound for the ith cycle, $[L_i, U_i]$, found in Algorithm 3.2.1 is an estimate for all $i > 0$. Therefore, combining this with the actual time difference for $i = 0$ results in an estimate of the worst-case time difference.

Theorem 3.2.1 *Algorithm 3.2.1 determines an estimate of the worst-case time difference between two events in a cyclic constraint graph for any cycle.*

Proof: In order to show that Algorithm 3.2.1 returns an estimate of the worst-case time difference, we must show that the following inequalities hold: $L' \leq L$ and $U' \geq U$ (from Definition 3.2.3). If $j > 1$ then Algorithm 3.2.1 returns $[L', U'] = [-\infty, \infty]$ which trivially satisfies Definition 3.2.3. If $j = 1$ then it returns $[L', U'] = [L_i, U_i]$. If $j = 0$ then Algorithm 3.2.1 returns $L' = \min(L_0, L_i)$ and $U' = \max(U_0, U_i)$. Since $[L_0, U_0]$ is an actual time difference for the 0th cycle, we only need to show that $[L_i, U_i]$ always yields an estimate for $i > 0$. A maximum time difference is calculated recursively in terms of other maximum time differences (see Algorithm 3.2.3). Therefore, when calculating U_i using subgraph G, one of two cases may occur. Its value may be independent of *maxdiff* values for events not in graph G (i.e., events from cycles less than $i-1$). If this is the case, then $U_i = \min_{i \geq 1}\{U_i\}$. On the other hand, if it depends on time differences of earlier events not in graph G, then just before *MaxDiff* falls off the end of the graph, it calls either $MaxDiff(G, source, \langle f, i-1\rangle)$ (1) or $MaxDiff(G, \langle f, i-1\rangle, source)$ (2). Since the rule between $\langle f, i-1\rangle$ and *source* has timing constraint $[0, \infty]$, (1) will return ∞, and (2) will return 0. If graph G were extended to include another cycle, the rule between *source* and $\langle f, i-1\rangle$ would be replaced with a rule

of the form $\langle e, f, l, u\rangle$. Now, *MaxDiff*$(G, \langle e, i-2\rangle, \langle f, i-1\rangle)$ would be called which would return a value less than or equal to ∞, or *MaxDiff*$(G, \langle f, i-1\rangle, \langle e, i-2\rangle)$ would be called which would return a value less than or equal to 0 (note this second case is never positive because from the ordering defined by the rule, we know that e always occurs before f). This relationship continues to hold if the graph is extended an infinite number of cycles. Since the value found for case (1) and for case (2) is greater than that found if graph G is extended back further, and since the maximum time difference is calculated by adding these values to values found on the rest of the graph, we know that the value calculated for U_i using graph G will be less than or equal to the actual value of U_i for $i > 1$. Therefore, $U' \geq U$, and we can similarly show that $L' \leq L$. Thus, Algorithm 3.2.1 gives an estimate of the worst-case time difference. ∎

3.2.4 Complexity of the Algorithm

Calculating the time difference of each pair of events using the *MaxDiff* algorithm has complexity $O(v \cdot e)$ where v is the number of vertices and e is the number of arcs in the graph [46]. Let $|E'|$ and $|R'|$ be the number of events and rules, respectively, in the cyclic constraint graph representation. The largest graph which Algorithm 3.2.1 analyzes has $2|E'|$ vertices and $2|R'|$ arcs. Therefore, using Algorithm 3.2.1 to calculate estimates for all time differences has complexity $O(|E'| \cdot |R'|)$.

3.2.5 Extensions to Find a Better Estimate

If either the bound is not tight enough or there is interest in finding worst-case time differences of events across more than one cycle (i.e., $j > 1$), the algorithm can be extended by increasing the size of the subgraphs which Algorithm 3.2.1 analyzes. Assuming subgraph G is enlarged to contain c cycles ($c = 2$ in Algorithm 3.2.1), the algorithm is modified in the following ways:

1. Construct subgraph G using only events from cycles $i-(c-1), \ldots, i$ where $i > (c-2)$.

2. Construct subgraph G' using only events from cycles $i \leq (c-2)$.

3. If $j \leq (c-2)$ then using graph G', find $[L_j, U_j], \ldots, [L_{(c-2)}, U_{(c-2)}]$.

4. $L' = \min(L_i, L_j, \ldots, L_{(c-2)})$ and $U' = \max(U_i, U_j, \ldots, U_{(c-2)})$.

In the modified algorithm, estimates of worst-case time differences with $j \leq (c-1)$ can now be found. Theorem 3.2.1 can easily be extended to show that the modified algorithm returns an estimate of the worst-case time difference. It is also easy to show that the complexity of the modified algorithm is $O(c|E'| \cdot c|R'|)$.

3.2.6 Termination of the Algorithm

In order to avoid unnecessary calculations, the algorithm can be modified to check if extending the size of the subgraphs analyzed (i.e., increasing c) is helpful. To do this, the algorithm is modified to return a best-case estimate, $[L_{best}, U_{best}]$, in addition to the worst-case estimate, $[L', U']$, where $L_{best} = \min(L_j, \ldots, L_{(c-2)})$ and $U_{best} = \max(U_j, \ldots, U_{(c-2)})$. Given the actual worst-case time difference is $[L, U]$, it is easily shown that these estimates satisfy the inequalities: $L' \leq L \leq L_{best}$ and $U_{best} \leq U \leq U'$. If tightening the bound to $[L_{best}, U_{best}]$ would not result in less circuitry than $[L', U']$, then it is not worth increasing c. In fact, if $L_{best} = L'$ and $U_{best} = U'$, then the actual worst-case time difference $[L, U]$ has been found. In general, increasing c does not guarantee that the exact bound $[L, U]$ can always be found, but in all the circuit examples that we synthesized using Algorithm 3.2.1 (i.e., $c = 2$), it either found the exact bound or at least a sufficiently tight bound to detect all redundancies.

3.2.7 Removing Redundant Rules

One application of this timing analysis algorithm is to determine if a rule in a conflict-free timed ER structure is *redundant*. A rule is redundant in a timed ER structure if its omission does not change the behavior specified. In other words, given a structure S and a rule r, a new structure S' constructed by removing r from its rule set has the same set of timed configurations (i.e., $\mathcal{C}(S') = \mathcal{C}(S)$).

If there are multiple rules enabling an event, then it is possible that some of them are redundant. In addition to the redundant rules described in Section 3.1, timing analysis can be used to find additional redundant rules. Algorithm 3.2.4 checks each rule by using Algorithm 3.2.1 to find an estimate of the worst-case time difference between the enabled and enabling event. If the lower bound of this estimate is larger than the upper bound of the timing constraint on the rule, then this rule cannot be constraining the behavior so it is redundant.

Algorithm 3.2.4 (Find redundant rules)
structure FindRed(timed ER structure $\langle A_0, E_0, R_0, R'_0 \rangle$*) {*
foreach *rule of the form* $\langle e, f, l, u \rangle$ *in* R_0 {
$[L', U']$=*WCTimeDiff(* $\langle A_0, E_0, R_0, R'_0 \rangle, e, f, 0$*);*
If ($L' > u$*) then* $R_0 = R_0 - \{\langle e, f, l, u \rangle\}$*;*
}
foreach *rule of the form* $\langle e, f, l, u \rangle$ *in* R'_0 {
$[L', U']$=*WCTimeDiff(* $\langle A_0, E_0, R_0, R'_0 \rangle, e, f, 1$*);*
If ($L' > u$*) then* $R'_0 = R'_0 - \{\langle e, f, l, u \rangle\}$*;*
}
return($\langle A_0, E_0, R_0, R'_0 \rangle$);
}

Figure 17: Algorithm to find redundant rules.

The SCSI protocol controller depicted in Figure 12 has four events that are enabled by multiple rules: $req \downarrow$, $rdy \downarrow$, $req \uparrow$, and $q \uparrow$. For the rule, $\langle q \downarrow, rdy \downarrow, 0, 5 \rangle$, Algorithm 3.2.1 estimates the worst-case time difference between the two events $rdy \downarrow$ and $q \downarrow$ to be the bound $[15, 55]$. Since the lower bound of this time difference, 15, is greater than the upper bound of the timing constraint on the rule, 5, the rule is found to be redundant. In other words, the rule between the events $q \downarrow$ and $rdy \downarrow$ can be removed without changing the specified behavior. Further analysis finds this to be the only redundant rule.

3.3 Orbital Nets

Algorithm 3.2.1 is limited to analyzing conflict-free timed ER structures, and therefore, it can only be used on deterministic specifications. In order to address non-deterministic timed ER structures, we use partial order timing, an efficient, general algorithm which operates on an orbital net representation to find the reachable state space. In this section, we describe the orbital net representation, and how to translate a timed ER structure into an orbital net which satisfies the necessary properties to be analyzed using partial order timing analysis.

An orbital net is essentially a labeled safe Petri net extended with automatic net constructions and syntactic shorthands. The net constructions allow us to have relatively straightforward operational semantics, while the syntactic shorthands allow us to compose

the nets without an exponential blowup in net size. These features are described in detail in [60]. Orbital nets also include constructs for specifying timing requirements and *simultaneous* actions which allow us to easily mix behavior and environmental requirements even at the gate model level. These last two features are described in detail in the following subsections.

An orbital net is modeled by the tuple $\langle A, P, T, F, M_0, TR, L\rangle$ where A is the set of atomic actions, P is the set of places, T is the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of edges, $M_0 \subseteq P$ is the initial marking, TR is an assignment of timing requirements to places, and L is a function which labels transitions with sets of simultaneous actions. For a place $p \in P$, the *preset* of p (denoted $\bullet p$) is the set of transitions connected to p (i.e., $\bullet p = \{t \in T \mid (t,p) \in F\}$), and the *postset* of p (denoted $p\bullet$) is the set of transitions to which p is connected (i.e., $p\bullet = \{t \in T \mid (p,t) \in F\}$). For a transition $t \in T$, the presets and postsets are similarly defined (i.e., $\bullet t = \{p \in P \mid (p,t) \in F\}$ and $t\bullet = \{p \in P \mid (t,p) \in F\}$).

3.3.1 Timing Requirements

Timing in an orbital net is associated with a place as a timing requirement consisting of a lower bound, an upper bound, and a type (denoted $\langle l, u\rangle type$). The lower bound is a nonnegative integer and the upper bound is an integer greater than or equal to the lower bound, or ∞. Again, since real values can be expressed as rationals within any required accuracy, restricting the bounds of timing requirements to be integers does not decrease the expressiveness of orbital nets.

There are two types of timing requirements: *behavior (b)* and *constraint (c)*. Behavior timing requirements are used to specify guaranteed timing behavior. Constraint timing requirements, on the other hand, are used to specify desired timing behavior, and they do not affect the actual timing behavior. If the timing requirement on a place is omitted, it is assumed to be $\langle 0, \infty\rangle c$.

Consider a D-type flip-flop (FF) pictured in Figure 18(a). The timing requirements for the FF are depicted using a timing diagram in Figure 18(b) and using an orbital net in Figure 18(c). This FF has a *setup time* of 5 time units which is represented with a constraint timing requirement from the rising transition on the input D to the rising transition on the clock φ. Similarly, a *hold time* of 5 time units is represented with a constraint timing requirement from the rising transition on the clock φ to the falling transition on the input D. Note that these are requirements that the environment must satisfy, and the FF cannot

guarantee this behavior. The delay of the FF is represented as a behavior timing requirement from the rising transition of the clock φ to the rising transition on the output Q. This requirement says that the FF circuit will generate $Q\uparrow$ between 5 and 8 time units after $\varphi\uparrow$.

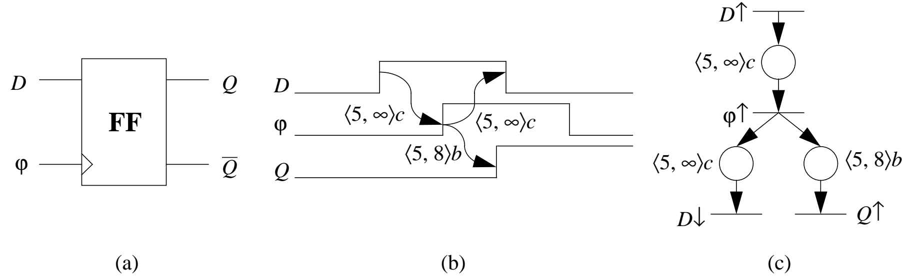


Figure 18: (a)A D-type flip-flop; (b) its timing requirements represented using a timing diagram; (c) its timing requirements represented using an orbital net.

When there is a single behavior place p in the preset of a transition, regardless of the interpretation, the time of occurrence of a transition in the postset of p (denoted $t(p\bullet)$) is always greater than the time of occurrence of any transition in the preset of the place (denoted $t(\bullet p)$) by at least the lower bound of the timing requirement on p, and it is always less then the upper bound. If, on the other hand, there are multiple behavior places in the preset of a transition, there are four different ways the specified behavior can be interpreted [72]. The first, or *type 1*, says that for all behavior places p, $t(p\bullet) - t(\bullet p)$ must exceed the lower bound but must not exceed the upper bound (this is the type used by our constraint places). If no possible timing behavior satisfy these requirements, the specification is inconsistent. The second, or *type 2*, says that for all behavior places p, $t(p\bullet) - t(\bullet p)$ must exceed the lower bound and for at least one behavior place, $t(p\bullet) - t(\bullet p)$ must not exceed the upper bound. This is the type usually associated with circuit behavior, so it is the type we associate with our behavior places. *Types 3* and *4* are duals in which only a single lower bound needs to be reached (i.e., an OR relationship). These two types are not considered as they do not correspond with the conjunctive nature of the Petri-net model.

The partial order timing analysis algorithm described later relies on the fact that each behavior place represents a single nondeterministic choice of delay that cannot be affected by other behavior places. When there are multiple behavior places in the preset of a transition, the type 2 semantics allow the delay between the transition in the preset and

postset of a behavior place to exceed the requirements upper bound if the transition in the postset is being constrained by another behavior place. Therefore, the partial order timing analysis algorithm requires specifications to include at most a single behavior place in the preset of each transition. Fortunately, the original orbital net specification can always be transformed, as described later, into one which satisfies this *single behavior place requirement*.

3.3.2 Simultaneous Actions

For a large class of speed-independent and delay-insensitive designs, any hazard is potentially fatal [5], so simple delay models that are easy to integrate into gate models suffice. With the more complex delay models required for modeling real-time circuit delay, such integration is no longer easy or straightforward. Labeling each transition in an orbital net with a (possibly empty) set of simultaneous actions remedies this difficulty by allowing the function of a gate to be modeled separate from its delay behavior without a significant blowup in the state space size.

Consider, for example, an AND gate with a delay of 2 to 4 time units. Under the *output delay model*, the gate is modeled with an instantaneous function block followed by a delay element as shown in Figure 19(a). The orbital net corresponding to the functional behavior of the AND gate is given in Figure 19(b). In this net, there are four places corresponding to the four states of the two input signals a and b, and the value of c in each place tracks exactly the AND of the signals a and b. The orbital net corresponding to a simple delay element is shown in Figure 19(c). The behavior place labeled $\langle 2, 4 \rangle$ indicates that an output will occur between 2 and 4 time units after the preceding input occurs; no behavior violating this requirement will be generated by the net. The constraint places do not constrain the behavior of the net, but if another input event occurs before the preceding output event then the environment violates the specification. Composition of these nets gives an AND gate operating under the output delay model. In a similar manner, an AND gate operating under the *input delay model* could also be obtained.

The delay model shown in Figure 19(c) is relatively simple, and it suffices for many types of circuits. More complex delay models can and have been constructed, modeling more accurately the behavior of a gate under hazard conditions; for these, the separation of gate models into combinational function and delay behavior is essential [60].

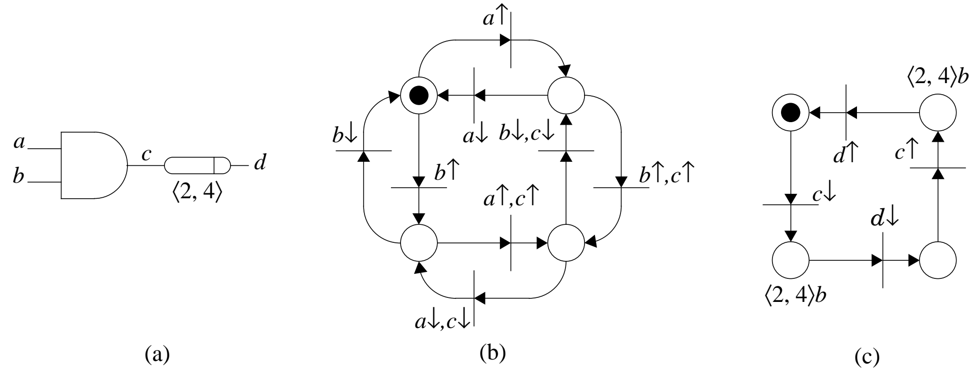


Figure 19: (a) AND gate with inputs a and b, and output d; (b) orbital net for its functional behavior; (c) delay buffer with input c, output d, and delay of $\langle 2, 4 \rangle$.

3.3.3 Operational Semantics

The behavior specified by an orbital net that satisfies the single behavior place requirement is defined with an operational semantics composed of two types of operations: advancement of time and firing of transitions. In an orbital net, an *untimed state* is a marking of the net. A *timed state* is an untimed state with a time-valued clock clk_i associated with each marked place p_i. Each clock advances with time and denotes how long the place has been marked. Time is advanced by uniformly increasing these clocks by an amount τ which is less than or equal to *max-advance* for a given marking. The function *max-advance* is defined as the minimum difference over all marked behavior places between the upper bound of the timing requirement on the place and its clock, or ∞ if there are no marked behavior places. This upper limit on time advancement maintains the clocks for all behavior places below the maximum allowed by their range.

In an orbital net, a transition is *untimed-enabled* if all places in its preset are marked. A transition is *timed-enabled* when it is untimed-enabled and if there is a behavior place in its preset, this place's clock is greater than the lower bound of the timing requirement on the place. Any timed-enabled transition can be fired instantaneously, and any number of transitions can be fired without time advancing. A transition is fired by removing the marking in the places in its preset and discarding the clocks. The places in the postset of the fired transition are then marked, and all newly marked places are assigned a clock initialized to zero.

Before firing a transition, however, the constraint places in the entire net must be

checked, and if any constraint place p_i is marked with $clk_i > u_i$, this firing is marked as a failure. Also, the clocks corresponding to a marking that is removed from a constraint place p_i must be checked, and if $clk_i < l_i$, this firing is also marked as a failure. Finally, after the firing of a transition, every marked behavior place must have a transition in its postset that is untimed-enabled in the new state; if this condition is not satisfied, this firing is a failure. This requirement ensures that every marked behavior place can fire in all states in which its timing conditions are met, and thus the value of its clock when it fires cannot be controlled by external state. If a failure is detected during synthesis, the specification is inconsistent and must be modified before an implementation can be obtained. If a failure is detected during verification, the timed circuit violates its specification.

These semantics define the set of *timed firing sequences* P, as a sequence of pairs of transition firings and time values. For simplicity, the time value represents a non-negative duration since the previous pair. Executing a timed firing sequence α on an orbital net results in the timed state $\mathit{fire}(\alpha)$. The set P is defined recursively. The empty sequence ε is in P. For every firing sequence α in P and for every value of τ such that $\tau \leq \mathit{max\text{-}advance}(\mathit{fire}(\alpha))$, then $\alpha(\phi, \tau)$ is in P, where ϕ represents an 'empty' firing. In addition, if a transition t is timed-enabled in $\mathit{fire}(\alpha)$, then $\alpha(t, 0)$ is also in P. The reachable state space is the range of the function *fire* over P.

3.3.4 Transformation from a Timed ER Structure to an Orbital Net

Winskel gave a construction from event structures to Petri nets [77]. We describe a similar construction from timed ER structures to orbital nets. Given a finite representation of a timed ER structure $S_0 = \langle A_0, E_0, R_0, \#_0, R_0', \#_0' \rangle$, Algorithm 3.3.1 constructs an orbital net $N = \langle A, P, T, F, M_0, TR, L \rangle$.

The algorithm first initializes the elements of the orbital net representation. The action set A for the orbital net is identical to the action set A_0 for the timed ER structure. Similarly, the transitions T in the net correspond to the events E_0 in the structure, and the labeling function L for the transitions in the net is the same as the labeling function L_0 for the events in the structure. The place set P, flow relation F, and initial marking M_0 are all initialized to the empty set. Finally, the next place label is set to 0.

Next, the algorithm translates each rule in the structure to connections in the orbital net. For each rule, a connection is added from the transition which corresponds to the enabling event to a place, and another connection is added from the place to the transition

which corresponds to the enabled event. The algorithm must first determine if a new place is going to be added to the net for these connections or if a place which already exists in the net should be used. A place is shared between multiple rules if either the enabling event conflicts with some other event which enables the same enabled event or the enabled event conflicts with some other event which is enabled by the same enabling event. The first four **if**-clauses check for either of these two conditions in the two rule sets R_0 and R_0'. Note the two rule sets are checked separately because it affects whether the place is in the initial marking or not. If there is no conflicting event or a place has not yet been added for the conflicting event, the last **if**-clause adds a new place to the orbital net with a timing requirement set by the timing constraint from the rule.

The function $place : E \times E \rightarrow P \cup \{none\}$ used in Algorithm 3.3.1 to find a place between two transitions is defined as follows:

$$place(e, f) \quad = \quad \textbf{if } \exists p.\{(e,p),(p,f)\} \subseteq F \textbf{ then } p \textbf{ else } none$$

If we apply these algorithms to the structures obtained for the *sel* process from the SEL, we obtain the orbital net shown in Figure 21(a). Part of the orbital net after composition with the other processes from the SEL is shown in Figure 21(b).

3.3.5 Satisfying the Single Behavior Place Requirement

Before the partial order timing analysis algorithm can be used, the orbital net must be transformed to one which satisfies the single behavior place requirement. To accomplish this, consider a fragment of an orbital net that has two behavior places in the preset of a transition shown in Figure 22(a). The desired timing behavior can be depicted graphically as shown in Figure 22(b). This net can be transformed to the one shown in Figure 23(a) which satisfies the single behavior place requirement. Basically, the idea behind this net transformation is that a path through the net is created for each possible ordering of the transitions in the preset. This has the effect that each transition in the preset is given the chance to be the last one preventing the transitions in the postset from occurring. For illustration purposes, additional events c_0 and c_1 are added to the net to occur simultaneously with the two transitions associated with c. The timing behavior of c_0 and c_1 are shown graphically in Figure 23(b) and (c), respectively. The behavior of these two together is exactly the desired timing behavior of c. For n behavior places, the net is transformed to model the $n!$ possible orderings of the n enabling events. While this transformation can lead to a

Algorithm 3.3.1 (Transform a timed ER Structure to an Orbital Net)

net struct2net(timed ER structure $\langle A_0, E_0, R_0, \#_0, R'_0, \#'_0 \rangle$*)* {
    $A = A_0$;
    $T = E_0$;
    $L(T) = L_0(E_0)$;
    $P = F = M_0 = \emptyset$;
    $nextp = 0$;
    foreach $\langle e, f, l, u \rangle \in R_0 \cup R'_0)$ {
        $p = none$;
        $initial = (\langle e, f, l, u \rangle \in R'_0)$;
        if $\exists g . e \# g \wedge \langle g, f, l, u \rangle \in R_0$ **then** {
            $initial =$ **false**;
            $p = place(g, f)$;
        }
        if $p = none \wedge \exists g . e \# g \wedge \langle g, f, l, u \rangle \in R'_0$ **then** $p = place(g, f)$;
        if $p = none \wedge \exists g . f \# g \wedge \langle e, g, l, u \rangle \in R_0$ **then** {
            $initial =$ **false**;
            $p = place(e, g)$;
        }
        if $p = none \wedge \exists g . f \# g \wedge \langle e, g, l, u \rangle \in R'_0$ **then** $p = place(e, g)$;
        if $p = none$ **then** {
            $p = nextp^{++}$;
            $P = P \cup \{p\}$;
            if $initial$ **then** $M_0 = M_0 \cup \{p\}$;
            $TR(p) = \langle l, u \rangle b$;
        }
        $F = F \cup \{(e, p), (p, f)\}$;
    }
    return $(\langle A, P, T, F, M_0, TR, L \rangle)$;
}

Figure 20: Algorithm to transform a timed ER structure to an orbital net.

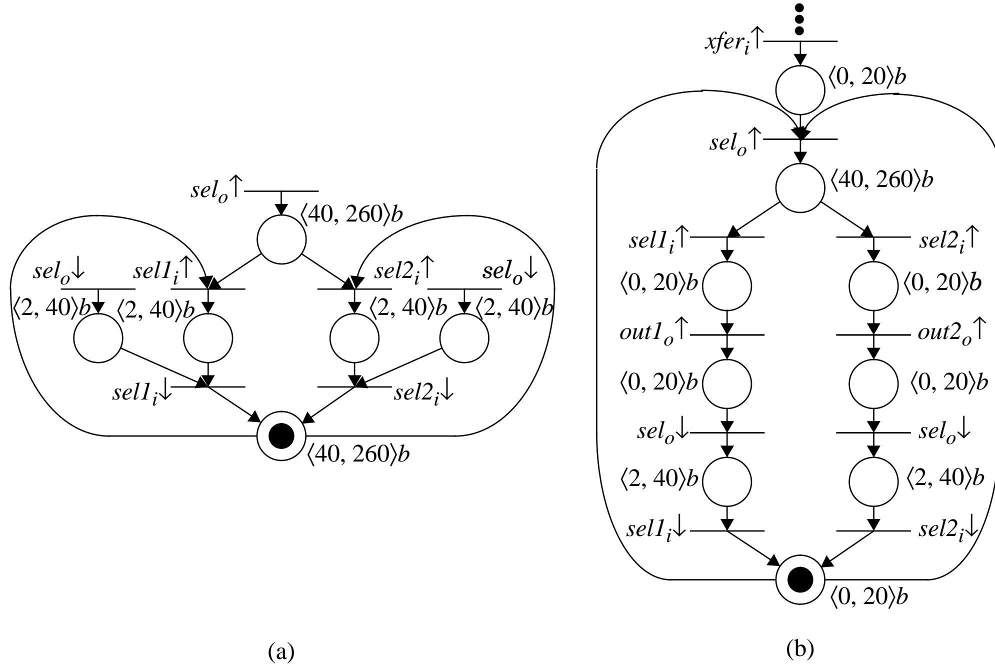


Figure 21: (a) Orbital net for the *sel* process from the SEL; (b) part of the orbital net after composition with the other processes.

substantial blowup in the net size, we have found that the value of n tends to be quite small in practical examples.

The transformation is more complicated in the case that one of the behavior places in the preset has multiple transitions in its postset. Consider a fragment of the orbital net from the SEL shown in Figure 24(a). In this net, the behavior place in the postset of $data_i \uparrow$ is shared by the transitions $out1_o \uparrow$ and $out2_o \uparrow$. In other words, if $out2_o \uparrow$ occurs, the marking is removed before it can contribute to the firing of $out1_o \uparrow$. In order to model this, the net is first transformed using the procedure described above for $out1_o \uparrow$ and $out2_o \uparrow$. Then, transitions are added to the part associated with $out1_o \uparrow$ on $out2_o \uparrow$ that reset the marking, and similarly transitions are added to the part associated with $out2_o \uparrow$ on $out1_o \uparrow$. A portion of the transformed net illustrating this is shown in Figure 24(b).

3.4 Partial Order Timing

In orbital nets, the clocks associated with each marking can take on real values, so there are an infinite number of timed states. In order to perform explicit state space exploration,

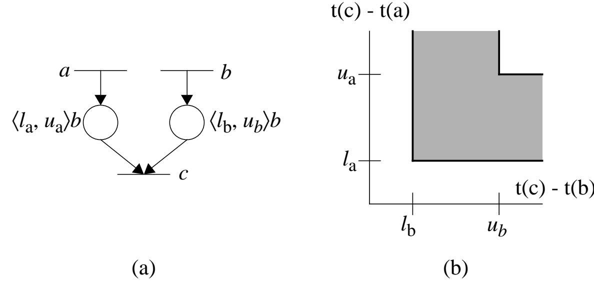


Figure 22: (a) Fragment of the orbital net that violates the single behavior place requirement; (b) graphical representation of the desired timing behavior.

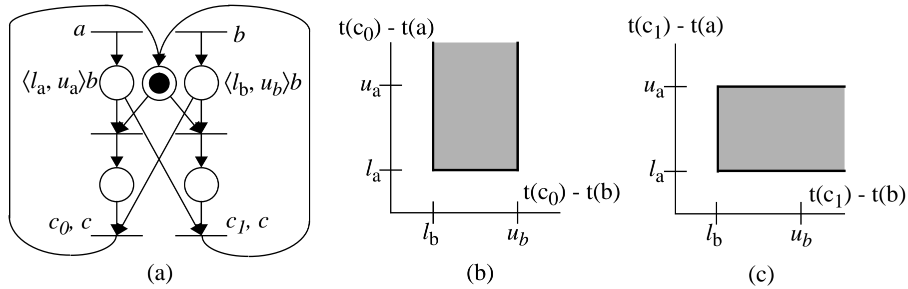


Figure 23: (a) Orbital net that satisfies the single behavior place requirement; graphical representation of the timing behavior of c_0 (b) and c_1 (c).

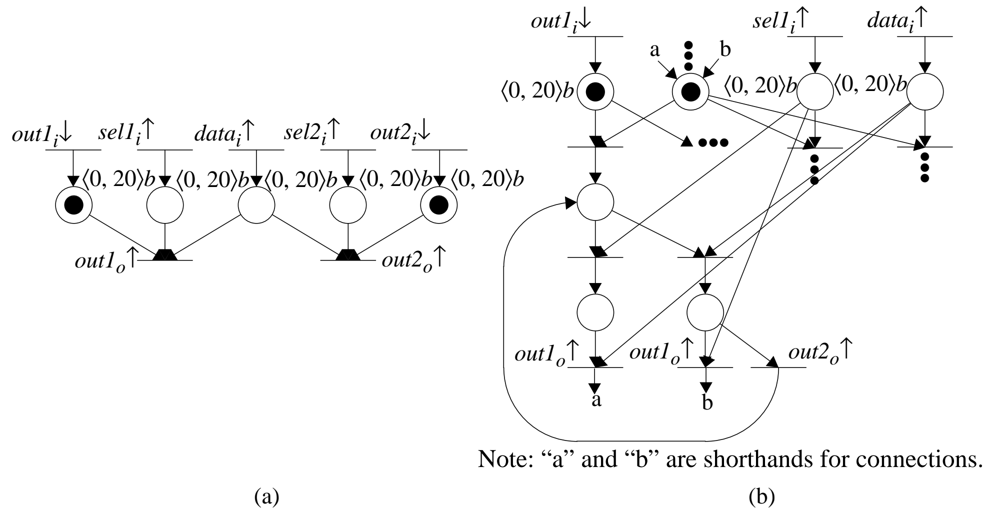


Figure 24: (a) Fragment of an orbital net with a behavior place that has multiple transitions in its postset; (b) part of the transformed orbital net that satisfies the single behavior place requirement.

we must either group the timed states into a finite number of equivalence classes or sets, or restrict the set of values that the clocks can attain.

Alur's unit-cube technique has the best known worst-case complexity for timed state space exploration of general timed systems [1]. This technique considers equivalence classes of timed states with the same integral clock values and a particular linear ordering of the fractional values of the clocks. For the case where there are two marked places and two clocks clk_1 and clk_2, the equivalence classes are pictured in Figure 25(a); every point, line segment, and interior triangle is an equivalence class. Let us assume the number of distinct untimed states in an orbital net is $|S|$. If the maximum value of any timing requirement is k, and there are at most n marked places in the net in any state (this value is trivially bounded by the size of the safe net), the worst-case size of the state space for his method is asymptotically [60],

$$|S| \frac{n!}{\ln 2} \left(\frac{k}{\ln 2}\right)^n 4^{1/k}.$$

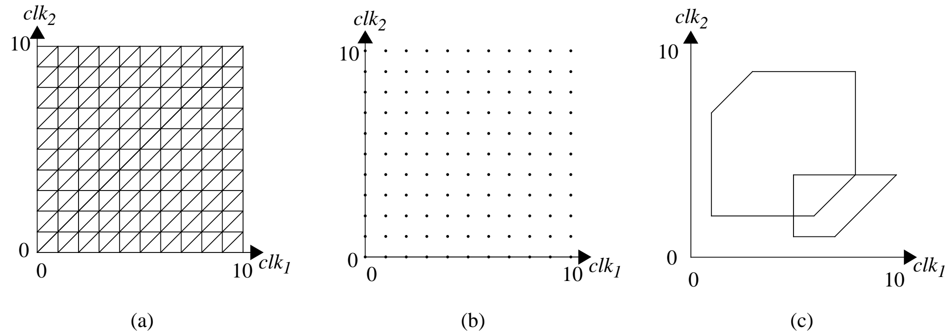


Figure 25: (a) Unit-cube, (b) discrete, and (c) geometric representations of the timed state space.

It has been proven, however, that the general unit-cube technique is unnecessary for orbital nets since considering only integer event times gives a full characterization of the continuous-time behavior [60] (this proof is similar to one given by Henzinger, et. al. in [32] for timed transition systems). In other words, only timed states associated with each discrete-time instance, represented as a point for the two-dimensional case in Figure 25(b), needs to be considered. This technique is used by Burch for verifying timed circuits [14], and as a worst-case state space size of $|S|\,(k+1)^n$ which is better than the unit-cube method by more than $n!$.

Both unit-cubes and discrete-time, however, are of little more than theoretical interest

because the size of the state space increases exponentially with the concurrency in the net. For a circuit with timing values accurate to two significant digits, with up to six independent concurrent pending events, the state space is easily in excess of 10^{12} states—well beyond the capabilities of most finite-state synthesis and verification techniques.

In this section, we first discuss *geometric timing*, a timing analysis technique that usually performs well in practice, even though the worst-case performance is much worse than either the unit-cube or the discrete-time approaches. Dill [23], Lewis [41], and Berthomieu and Diaz [8] originated geometric state space exploration, and it has become an active area of research [2, 31, 29]. Then, we describe our proposed technique, partial order timing, which improves upon the geometric methods by making use of concurrency and causality information. Recent work by Yoneda et. al. [79] also considered partial orders. Our work differs in that our formalism includes notions of specification, circuit composition, and receptiveness which enable us to perform efficient state space exploration on nontrivial timed circuit examples. To our knowledge, neither timed automata nor time Petri nets have been used in this fashion.

3.4.1 Geometric Regions

Rather than consider at each step a single discrete-time state, or a minimum equivalence class of timed states, the geometric timing method considers an infinite set of timed states in parallel. Specifically, convex geometric regions of timed states represented by upper and lower bounds on specific clock values and on the differences between pairs of specific clock values are used as the representation of the timed state space. The set of such constraints is usually represented by a matrix A, where the constraints on clocks $\{clk_1, \ldots, clk_n\}$ are of the form $clk_i - clk_j \leq a_{ji}$. A fictitious clock clk_0 that is always exactly zero is introduced so that upper and lower limits on a particular clock can be represented in the same form [23].

For any convex region that can be represented by such a matrix, there are many matrices that represent the same convex region. The process of *canonicalization* using Floyd's algorithm can be performed to yield a unique constraint matrix [23]. While in general Floyd's algorithm runs in time $O(n^3)$, since only incremental changes are made to the matrix during analysis, specializations of Floyd's algorithm that run in time $O(n^2)$ suffice [60]. Two sample regions are given in Figure 25(c).

3.4.2 State Space Exploration with Geometric Timing

Each geometric region can be considered as an infinite set of timed states which are operated on in parallel. In order to perform state space exploration using geometric timing, we redefine the operational semantics of orbital nets in terms of these geometric regions as opposed to individual timed states. We do not discuss the aspects of state space exploration that do not consider time, since they are the same in both cases. We describe how these operations work for a single step in a timed sequence, assuming it works for the predecessor sequence; the trivial base case and structural induction on sequences completes the proof that these operations work for all sequences.

In our original operational semantics, advancing time involves adding some number t to all clocks. For geometric regions, advancing time involves extruding the geometric region in the $clk_1 = clk_2 = \cdots = clk_n$ direction, subject to *max-advance*, which itself is a convex region.

Determining whether a particular transition is timed-enabled in our original operational semantics entails comparing the clocks with the timing requirements. With geometric regions, we determine the subset of the timed states in the region for which the particular transition is enabled. This can be performed by introducing the enabling conditions on the transition as additional constraints on the region and recanonicalizing. For orbital nets, these conditions describe a convex region in the appropriate form, and it is easy to show that the intersection of two such convex regions is a convex region of the same form. Canonicalization by definition does not reduce the set of timed states represented.

After selecting an enabled transition, firing that transition involves removing some set of clocks and introducing new clocks initialized to zero. With geometric regions, removing these clocks involves projection of the system of constraints to eliminate a particular set of variables, and introducing new clocks is done by adding a new set of variables equal to zero.

3.4.3 Performance of Geometric Timing

While unit-cubes and discrete-time operate on timed firing sequences, geometric timing operates over untimed firing sequences. The function *untime*(α) returns the underlying *untimed firing sequence* from a given timed firing sequence by stripping the timing and removing any ϕ firings. For each untimed firing sequence α operated on by geometric timing, it calculates directly the full set of timed states reachable from all timed firing

sequences β that satisfy $untime(\beta) = \alpha$. Thus, rather than separately considering every possible occurrence time for a particular transition in α during state space exploration, in one step the geometric region method considers all possible occurrence times.

State space exploration using geometric timing can be very efficient. However, some examples require an extremely large number of geometric regions such as the adverse example `adv4x40` shown in Figure 26. While only having a single untimed state, standard geometric timing techniques generate an incredible 219,977,777 distinct geometric regions. This is more than either the number of discrete-time states or unit-cube equivalence classes.

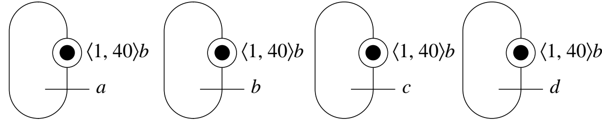


Figure 26: The adverse example `adv4x40` with $n = 4$ and $k = 40$.

3.4.4 Concurrency, Causality, and Posets

The major source of blowup in the adverse example is the way the standard geometric timing algorithm calculates exactly the set of timed states reachable from a sequence of transition firings; the transition firings are linearly ordered, even if they are concurrent in the system being evaluated. That is, if two concurrent transitions start clocks, the constraints between the two clocks reflect the linear order that the transitions are fired in the original sequence. For example, when the geometric timing algorithm analyzes the untimed firing sequence $[a, b]$, it obtains the upper geometric region shown in Figure 27, and when the algorithm considers the sequence $[b, a]$, it obtains the lower geometric region. In general, if there are n concurrent transitions that reset clocks visible in the resulting timed state, there are $n!$ different sequences that need to be considered, each of which leads to a distinct geometric region. For this reason, it is important to distinguish the causal ordering of transitions from the non-causal ordering that comes about from the selection of a particular firing sequence.

To solve this problem, we construct a partially ordered set, or *poset* for each untimed firing sequence which is represented with an acyclic, choice-free unfolding of the original orbital net. The poset reflects the causality and concurrency inherent in the firing sequence. Initially, the unfolded net representing the poset contains a single transition with places in its postset corresponding to each initially marked place. Transitions are added in the same

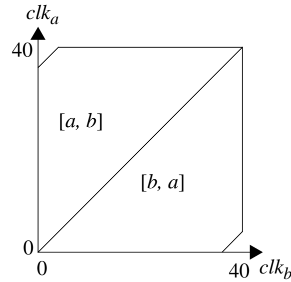


Figure 27: Geometric regions from the adverse example.

order as they occur in the firing sequence. For each transition in the firing sequence, a correspondingly labeled transition is added to the unfolded net. A set of arcs into the transition are connected from the most recently added places in the unfolded net corresponding to places in the preset of the transition in the original orbital net. Finally, a new set of places corresponding to the places in the postset of the transition in the original net are added, and these places are connected to the new transition. Every place and every transition in the unfolded net, except the first, correspond to some place and some transition in the original net. Every place and every transition in the original net correspond to zero or more places and transitions in the unfolded net.

A poset explicitly represents the concurrency in a particular firing sequence. That is, a particular poset corresponds to many different firing sequences that differ only in the interleavings of concurrent transitions; every such firing sequence fires the same set of transitions and leads to the same final untimed state. For example, the poset represented with the unfolded net shown in Figure 28 corresponds both to the sequence $[a, b]$ and to the sequence $[b, a]$.

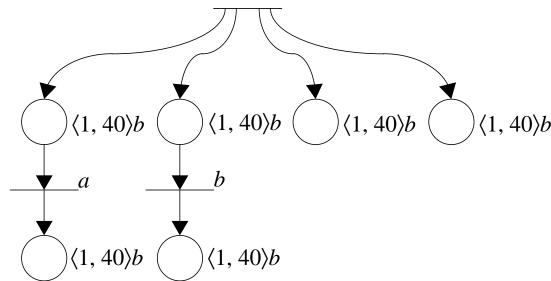


Figure 28: One poset from the adverse example.

3.4.5 State Space Exploration with Partial Order Timing

State space exploration proceeds just as it does for the previous methods based on sequences, except that, for each sequence, the algorithm constructs the corresponding unfolded net. With depth-first search, this is done incrementally. The algorithm also incrementally calculates a constraint matrix that stores the firing time relationship among the transitions. For each constraint place p, the constraint $t(\bullet p) \leq t(p\bullet)$ is introduced. For each behavior place p in the resulting unfolded net with a timing requirement of $\langle l, u\rangle b$, two constraints are introduced. The first reflects the minimum separation, $t(\bullet p) - t(p\bullet) \leq -l$. The second reflects the maximum separation, $t(p\bullet) - t(\bullet p) \leq u$. All constraints introduced in this fashion for a given unfolded net must be satisfied. After canonicalizing this constraint matrix, it has produced a geometric region that represents the full set of reachable states for the poset corresponding to the unfolded net. Applying this procedure to the unfolded net shown in Figure 28, we obtain at once the geometric region which encloses both regions shown in Figure 27.

While geometric timing operates on untimed firing sequences, partial order timing operates on posets. The function *poset* takes an untimed firing sequence and returns the corresponding unfolded net. For each untimed firing sequence α operated on by the partial order technique, it calculates directly the full set of timed states reachable from any timed firing sequence β such that $poset(untime(\beta)) = poset(\alpha)$. Thus, rather than separately considering every interleaving of concurrent transitions, in one step the partial order method considers all possible interleavings. For untimed state space exploration, different interleavings result in the same state. For timed state space exploration, different interleavings usually result in different sets of timed states, with different future behavior, leading to a combinatorial explosion of timed regions for each untimed state. Representing, as a single constraint matrix, the union of all timed states reachable from all possible interleavings, therefore, dramatically reduces the size of the state space representation. In fact, the partial order method typically reduces the average number of timed regions for each untimed state to a value close to one. For the adverse example in Figure 26, partial order timing obtains exactly one geometric region corresponding to the one untimed state.

3.4.6 Efficiency Considerations

The number of transitions in the unfolded net is equal to the length of the firing sequence plus one, and it increases with the depth of our search. Calculating the minimum separations between the occurrence times in the unfolded net, even with our incremental $O(n^2)$ approach, becomes prohibitively expensive as the firing sequence lengthens. In addition, the algorithm needs a constraint matrix for each step; this would require a tremendous amount of storage during depth-first search.

To keep n bounded as the depth of our search increases, the algorithm determines what prefix, if any, of the unfolded net can safely be ignored. The algorithm can eliminate any transitions that no longer affect future calculations. In general, the algorithm can eliminate a variable from any set of equations or inequalities whenever it has produced the full set of equations or inequalities that use that variable. Since all constraints introduced through the firing of a transition are associated with places connecting the new transition to the old, once a transition in the unfolded net no longer has any places in its postset which do not have a transition in their postset, it is eliminated from our constraint matrix. Thus, our n is—at most—the number of marked places in the original net at any given time, plus one for the current transition.

Because the number of geometric regions is typically small, a further optimization is possible. Rather than backtracking only when an identical geometric region is found, our search can backtrack whenever a new geometric region is a subset of a previously seen geometric region. Comparing two geometric regions for inclusion can be performed in $O(n^2)$ time.

3.5 Finding the Reduced State Graph

In order to generate a circuit implementation, many methodologies transform a higher-level specification into a *state graph* (SG) so that Boolean minimization techniques can be applied [17] [47]. Essentially, a state graph is a graph in which the vertices are bitvectors and the arcs are signal transitions. Each bitvector specifies the binary value of every signal in the system when the system is in that state. When synthesizing a timed circuit, one of the timing analysis algorithms described in this chapter is utilized to generate a *reduced state graph* (RSG) which often has significantly fewer states than a SG generated without considering timing constraints. Since the size of the SG and the complexity of the circuitry

are strongly correlated, our method often results in simpler circuitry compared with other methods that do not fully utilize timing constraints.

A RSG is a graph in which its vertices are untimed states and its edges are possible *state transitions*. A RSG is modeled by the tuple $\langle I, O, \Phi, \Gamma \rangle$ where I is the set of input signals, O is the set of output signals, Φ is the set of states, and $\Gamma \subseteq \Phi \times \Phi$ is the set of edges. For each untimed state s, there is a corresponding labeling function $s : I \cup O \rightarrow \{0, R, 1, F\}$ which returns the value of each signal and whether it is untimed-enabled, i.e.,

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

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

$$val(u) \equiv \begin{cases} 0 & \textit{if } u = 0 \textit{ or } u = R \\ 1 & \textit{if } u = 1 \textit{ or } u = F. \end{cases}$$

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

A state graph is defined to be well-formed if for any state transition (s, s') in Γ, the value of exactly one enabled signal in s changes to a new value in s'. A state transition (s, s') and the signal v that differs in value is denoted as follows: $s \xrightarrow{v} s'$. Our synthesis procedure also requires that the state graph be *complete state coded*, defined to be that if for any two states in which all signals have the same value, any output signal untimed-enabled in one state is also untimed-enabled in the other. It has been reported that adding state variables can transform an arbitrary state graph into one that satisfies complete state coding [19, 39, 74]. These approaches, however, may be conservative when timing is considered. Therefore, we believe adding state variables to a timed specification is an interesting open research problem.

If the timed ER structure for the timed HSE specification is conflict-free, Algorithm 3.2.1 can be used to derive the reduced state graph using a *constrained token flow* described in

Algorithm 3.5.1. This is similar to *token flow* which is used for finding state graphs as described in [47] [17]. The algorithm begins with the initial marking of the constraint graph which is defined as the set of rules enabled by *reset*. The function *FindState* is then used to find the state as defined above for the marking. Given a marking, an event f is enabled if all rules of the form $\langle e, f, l, u \rangle$ in both R_0 and R_0' are in the marking, or all rules of this form in R_0 and the rule $\langle reset, f, l, u \rangle$ is in the marking. If in a marking more than one event is enabled, all possible event sequences need to be generated. With timing constraints, it may be possible that one of the enabled events is always preceded by another. The function *Slow*, implemented in Algorithm 3.5.2, is used to check if an enabled event is slower than some other enabled event. If so, the occurrence of the slower event is postponed. The result is that some states are no longer reachable, yielding a reduced state graph. Note that if the function *Slow* is changed to always return *FALSE* then the resulting state graph is the same as generated using regular token flow.

Algorithm 3.5.1 (Find the reduced state graph)

```
set FindRSG(timed ER structure ⟨A_0, E_0, R_0, R_0'⟩) {
  initial_marking = {rules in R_0 of the form ⟨reset, f, l, u⟩};
  set_of_markings = {initial_marking};
  present_state = FindState(⟨A_0, E_0, R_0, R_0'⟩, initial_marking);
  set_of_states = {present_state};
  while (set_of_markings ≠ ∅) {
    take marking from set_of_markings
      (i.e., set_of_markings = set_of_markings − {marking} );
    foreach enabled event f in marking {
      if not (Slow(⟨A_0, E_0, R_0, R_0'⟩, f, marking)) then {
        new_marking = marking − {rules in marking of the form ⟨e, f, l, u⟩}
              + {rules in R_0 ∪ R_0' of the form ⟨f, g, l, u⟩};
        present_state = FindState(⟨A_0, E_0, R_0, R_0'⟩, new_marking);
        if (present_state ∉ set_of_states) then {
          set_of_states = set_of_states + {present_state};
          set_of_markings = set_of_markings + {new_marking};
  } } } }
  return(set_of_states);
}
```

Figure 29: Algorithm to find the reduced state graph.

Algorithm 3.5.2 (Check if an event is slow)

```
boolean Slow(timed ER structure ⟨A_0, E_0, R_0, R'_0⟩; event u; marking M) {
  foreach event v that is enabled in M where u ≠ v {
    j=FindCycleOffset(v,u);
    if (j ≥ 0) then {
      [L', U']=WCTimeDiff( ⟨A_0, E_0, R_0, R'_0⟩, u, v, j);
      if (U' < 0) then return(TRUE);
    } else {
      [L', U']=WCTimeDiff( ⟨A_0, E_0, R_0, R'_0⟩, v, u, (−1) ∗ j);
      if (L' > 0) then return(TRUE);
    }
  }
  return(FALSE);
}
```

Figure 30: Algorithm to check if an event is slow.

Using this algorithm on the SCSI protocol controller with the function *Slow* replaced with *FALSE* (i.e., ignoring the timing constraints), the SG obtained contains 20 states as shown in Figure 31(a). If the timing constraints are considered, a RSG is derived which contains 16 states as shown in Figure 31(b).

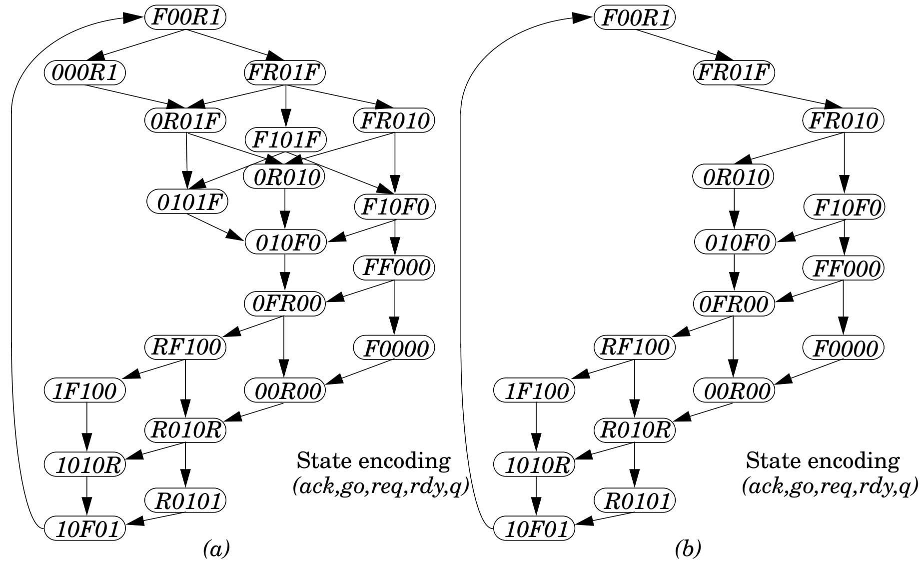


Figure 31: (a) SG and (b) RSG for the SCSI protocol controller.

The SEL introduced in Chapter 2 has non-deterministic behavior, namely input choice, so its timed ER structure is not conflict-free and the RSG cannot be found using Algorithm 3.5.1. Instead, the timed ER structure for the SEL is converted to an orbital net using Algorithm 3.3.1 which is further transformed to satisfy the single behavior place requirement. The resulting net is then analyzed using the partial order timing analysis algorithm to find the reduced state graph. The resulting RSG for the SEL contains 53 states. A SG generated ignoring all the timing information contains 256 states. As shown in the next chapter, the smaller RSG produces a significantly smaller circuit implementation.

Appendix

The usage of these timing analysis algorithms within ATACS is described in this appendix. After a timed HSE specification and has been compiled to a timed ER structure using the command *compile* and/or loaded using the command *loader*, several checks are done to make sure the timed ER structure is well-formed. The first is a *liveness* test which checks that every cycle has at least one arc which is initially marked. The second is a *connectivity* test which checks that the graph is strongly connected. If the graph is not strongly connected, the command *connect* can be used to attempt to add rules to make it strongly connected. The third check is a *safety* test which checks that every event exists in a cycle that includes just one initially marked arc. If any check fails, an error report can be obtained while in *verbose* mode to help track down the cause of the error.

After obtaining a well-formed timed ER structure, the program then attempts to eliminate any redundant rules. If the program detects that the timed ER structure is conflict-free, it uses Algorithm 3.2.4 to remove redundant rules. In either case, rules which have alternative paths that make them redundant as described in Section 3.1 are removed. In verbose mode, a list of redundant rules is stored to the file named ⟨filename⟩.rr. An example of such a file is shown in Figure 32 for the SCSI protocol controller.

< go+/1,go-/1,0,[20,50] >
< go-/1,go+/1,1,[20,50] >
< ack+/1,ack-/1,1,[20,50] >
< ack-/1,ack+/1,0,[20,50] >
< q-/1,rdy-/1,0,[0,5] >

Figure 32: Redundant rules from the SCSI protocol controller.

The program is now ready to find the RSG. If the program detects that the timed ER structure is conflict-free, it uses Algorithm 3.5.1. Otherwise, the timed ER structure is converted to an orbital net using Algorithm 3.3.1 which is further transformed to satisfy the single behavior place requirement. The resulting net is then sent to the program ORBITS written by Tom Rokicki [60] to apply the partial order timing analysis algorithm to find the RSG. The resulting RSG is then read back into ATACS for the rest of the synthesis procedure as described in the next chapter. In verbose mode, the RSG is stored to the file named ⟨filename⟩.rsg. An example of such a file is shown in Figure 33 for the SCSI protocol controller's reduced state graph depicted in Figure 31(b).

```
SG:
STATEVECTOR:INP go INP ack q req rdy
STATES:
011F0
0F10R
RFF01
RF001
1F00F
R0001
1000F
F00R0
000R0
FR010
0RR10
F1010
01R10
0R110
FF000
0F000
```

Figure 33: Reduced state graph for the SCSI protocol controller.

There are several commands which are related to timing analysis. First, the command *si* sets the timing constraints on all rules to $\langle 0, \infty \rangle$ and turns off all timing analysis, so as to produce speed-independent designs. The command *cycles* $\langle$*number*$\rangle$ can be used to change the number of cycles that the graph is unrolled when finding the worst-case time differences. The command *findtd* finds all time differences for the current number of cycles and stores them to the file named ⟨filename⟩.td. Similarly, the command *findwctd* finds all estimates of the worst-case time difference and stores them to the file named ⟨filename⟩.wctd. The commands *shower* and *printer* display and print the cyclic constraint graph or orbital net for the current design. Similarly, the commands *showrsg* and *printrsg* display and print the reduced state graph for the current design. The command *storeer* can be used to store the current timed ER structure to a file after modifications have been made to it such as removing redundant rules. Finally, it is possible to load a reduced state graph directly for synthesis in the form depicted in Figure 33 using the command *loadrsg*.

Chapter 4

Synthesis

...shifts up and down, everybody knows its wrong
—skinny puppy

Synthesis is the process of transforming a specification into a circuit implementation. Our synthesis procedure begins with a RSG representation derived using the timing analysis algorithms described in the previous chapter from which a hazard-free timed circuit implementation is generated using only basic gates such as AND gates, OR gates, and C-elements. From a RSG there are several different approaches that could be used to obtain a gate-level timed circuit implementation. The first approach is to use a traditional boolean minimization technique directly. We demonstrate, however, that when mapping the resulting implementation to basic gates, it may result in a hazardous implementation. Another approach is to split the design of the rising and falling transitions to obtain a *generalized C-element* implementation [44] and decompose it to basic gates. This technique alleviates some of the hazard problems, but we demonstrate that it may still be hazardous when mapped to basic gates. We take a *standard C-implementation* approach in which each rising and falling region for each output signal is implemented using a single *cube*, or AND gate, which must satisfy certain correctness constraints. A covering problem is setup and solved to find an optimal implementation for each region. When all the regions are merged, the resulting timed circuit implementation is guaranteed to be a hazard-free at the gate-level.

4.1 Sum-of-Products Implementation

After obtaining a RSG, we could apply a traditional Boolean minimization technique to find an implementation. Using this technique, the state space is partitioned into an *on-set*, an *off-set*, and a *don't-care-set*. Then, a Boolean minimization program, such as `espresso` [10] can be used to find the optimal sum-of-products representation. For our designs, a minimization problem would be setup for each output signal u with the on-set containing each state s in which the signal is enabled to rise or is stable high (i.e., $s(u) = R$ or $s(u) = 1$), the off-set containing each state s in which the signal is enabled to fall or is stable low (i.e., $s(u) = F$ or $s(u) = 0$), and the don't-care-set containing all unreachable states (i.e., $\beta^{I \cup O} - \Phi$).

Applying this technique to the signal $out2_o$ from the SEL results in the Boolean equation:

$$out2_o = (data_i \wedge sel2_i \wedge \neg out2_i) \vee (data_o \wedge out2_o) \vee (\neg xfer_o \wedge out2_o)$$

In order to guarantee correctness, Chu [17] and Meng [47] assumed that the logic equation for each output signal could be implemented directly with a single complex *atomic* gate. In other words, each signal is built with an instantaneous function block with a delay element connected to its output. Unfortunately, if the equation is mapped to basic gates and the delays of these gates are considered individually, the implementation may be hazardous. For example, the equation for $out2_o$ could be implemented directly as a sum-of-products as shown in Figure 34. If the 3-input AND and OR gates (gates 1 and 4) are assumed to have a delay of $\langle 2, 5 \rangle$ while the 2-input AND gates (gates 2 and 3) have a delay of $\langle 2, 3 \rangle$, this implementation is hazard-free. However, if the upper bound of the delay on the 2-input AND gates increases to 4 or more time units, this circuit is now hazardous. The segment of the state graph to the left illustrates a sequence of transitions which cause a hazard. Essentially, after gate 1 has caused $out2_o$ to rise, it has the potential of being shut off again before gates 2 or 3 can come on to hold the state.

In order to solve this problem, Lavagno [38] first mapped the logic equations ignoring hazards using standard synchronous techniques, then added delay elements where necessary to remove any potential hazards. This technique, however, not only adds additional overhead in terms of area and delay, but the resulting circuits may not be very reliable due to the difficulty in designing delay elements with accurate timing.

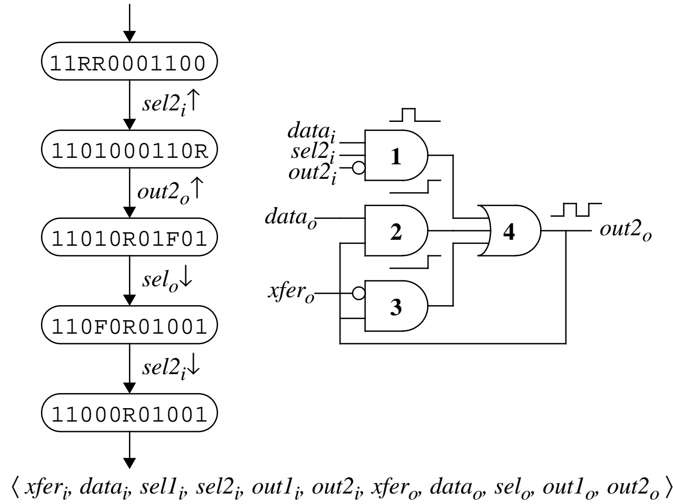


Figure 34: A hazardous sum-of-products implementation of $out2_o$ from the SEL.

4.2 Generalized C-Implementation

Another implementation strategy originally proposed by Martin [44] is to use generalized C-elements as the basic building blocks. This is also the technique used in our earlier work [52]. In this technique, the implementation of the set and reset of a signal are decoupled. The basic structure is depicted in Figure 35(a) in which the upper sum-of-products represents the logic for the set, the lower sum-of-products represents the logic for the reset, and the result is merged with a C-element. This can be implemented directly in CMOS as a single compact gate with weak-feedback as shown in Figure 35(b) or as a fully-static gate as shown in Figure 35(c).

Using a procedure similar to the one described in [52], we obtain a generalized C-element implementation for the signal $out2_o$ shown in Figure 36. While this could be implemented with a single generalized C-element, a gate-level implementation would be composed of a 3-input AND gate, a 2-input AND gate, and a C-element. Although this implementation no longer has the hazard associated with the RSG fragment in Figure 34, it now has a hazard illustrated with the state graph shown to the left in which the reset AND gate glitches while the output is stable low. For the specified delays, it can be shown that the hazard does not propagate to the output, but given appropriate delays this hazard may propagate [5]. To address this problem, after a generalized C-element implementation is produced and decomposed to basic gates, the design could be back-annotated with delays from the gate

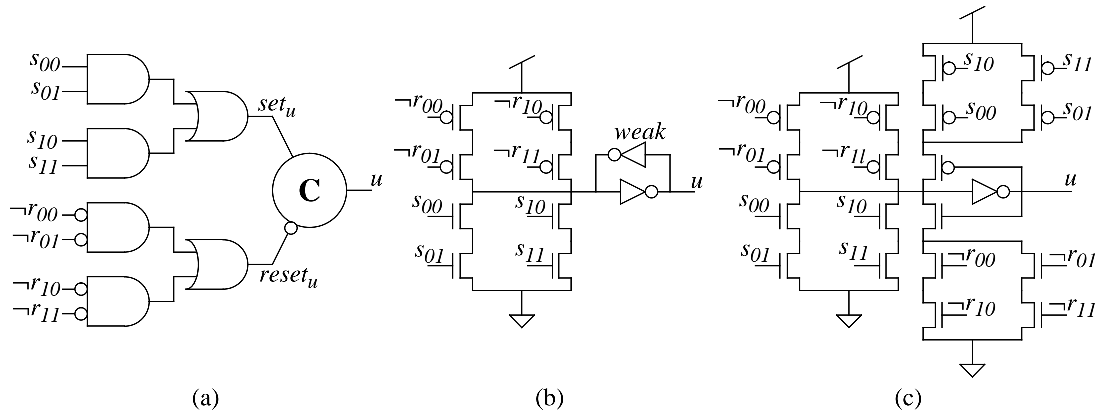


Figure 35: (a) The generalized C-element configuration with (b) weak-feedback and (c) fully-static CMOS implementations.

library, and the circuit could be verified. While this may often work, it is not clear what to do in the cases in which a hazard does exist. Also, a hazard is a spurious transition which wastes power and does no useful work. In a power efficient implementation, it is desirable to have logic which is hazard-free both internally and externally.

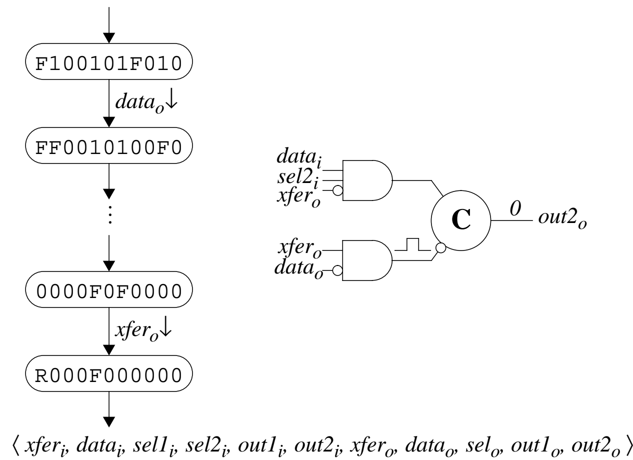


Figure 36: A hazardous gate-level implementation of $out2_o$ from the SEL.

4.3 Standard C-Implementation

To avoid the hazard concerns discussed above, our approach obtains a gate-level implementation by first decomposing the design into a set of rising and falling regions which are each implemented using a single cube. While the general structure of the standard C-implementation shown in Figure 37 is similar to the generalized C-element structure, each cube in the set or reset block must satisfy certain constraints to guarantee that the merged implementation is a gate-level hazard-free circuit. The approach is conservative in that timing analysis may show that the decomposed generalized C-element implementation is sufficient, but the overhead required tends to be small to get a safe implementation that is free of internal hazards.

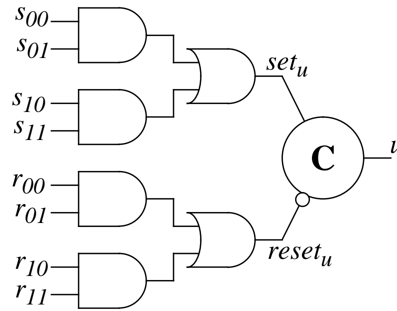


Figure 37: The standard C-implementation.

4.3.1 Excitation Regions and Quiescent States

In order to obtain a standard C-implementation, the RSG is decomposed for each output signal into a collection of *excitation regions*. An excitation region for the output signal u is a maximally connected set of states in which the signal is enabled to change to a given value (i.e., $s(u) = R$ or $s(u) = F$). If the signal is rising in the region (i.e., $s(u) = R$), it is called a *set region*, and the kth set region for a signal u is denoted $ER(u\uparrow, k)$. Similarly, if the signal is falling in the region (i.e., $s(u) = F$), it is called a *reset region*, and it is denoted $ER(u\downarrow, k)$. Typically, different excitation regions correspond to different output signal transitions in a high-level specification. For example, there are two set regions for the signal $xfer_o$ in the SEL which correspond to the two instances of $xfer_o\uparrow$ in the timed HSE specification in Figure 5.

For each signal u, there are two sets of stable, or *quiescent states*. There is the set of states where the signal u is stable high denoted $QS(u\uparrow)$ (i.e., $QS(u\uparrow) = \{s \in \Phi \mid s(u) = 1\}$), and the set where it is stable low denoted $QS(u\downarrow)$ (i.e., $QS(u\downarrow) = \{s \in \Phi \mid s(u) = 0\}$).

4.3.2 Correct Covers

We assume each excitation region will be implemented with a single AND gate, or *cube*, corresponding to a *cover* of the excitation region. The cover of a set region $C(u\uparrow, k)$ (or a reset region $C(u\downarrow, k)$) is a set of states for which the corresponding cube in the implementation evaluates to one. In order for a cover to lead to a hazard-free implementation, it must satisfy certain *correctness constraints* [7, 54]. These constraints guarantee that any gate in the implementation only changes when it is actively driving the output signal to change. This ensures that the transition of the gate is *acknowledged*.

First, a correct cover needs to satisfy a *covering constraint* which says that the reachable states in the cover must include the entire excitation region but must not include any states outside the union of the excitation region and associated quiescent states, i.e.,

$$ER(u*, k) \subseteq [C(u*, k) \cap \Phi] \subseteq [ER(u*, k) \cup Q(u*)]$$

where "*" indicates either "↑" for set regions or "↓" for reset regions.

Second, the covers of each excitation region must also satisfy an *entrance constraint* to ensure hazard-freedom. This constraint says that the cover must only be entered through excitation region states, i.e.,

$$[(s, s') \in \Gamma \wedge s \notin C(u*, k) \wedge s' \in C(u*, k)] \Rightarrow s' \in ER(u*, k)$$

To optimize the implementation, a single AND gate can be allowed to implement multiple regions. First, the procedure finds AND gate covers for each excitation region using modified correctness constraints. The covering constraint is modified to allow the cover to include states from other excitation regions, i.e.,

$$ER(u*, k) \subseteq [C(u*, k) \cap \Phi] \subseteq \left[\bigcup_l ER(u*, l) \cup Q(u*)\right]$$

The entrance constraint is similarly modified to allow the cover to be entered from any corresponding excitation region state, i.e.,

$$[(s, s') \in \Gamma \wedge s \notin C(u*, k) \wedge s' \in C(u*, k)] \Rightarrow s' \in \bigcup_l ER(u*, l)$$

An additional constraint is also now necessary to guarantee that an AND gate either covers all of an excitation region or none of it, i.e.,

$$ER(u*, l) \not\subseteq C(u*, k) \Rightarrow ER(u*, l) \cap C(u*, k) = \emptyset$$

Second, after the covers are found for each excitation region, a disjoint set of these covers must be selected to cover all regions. It is possible that no such set exists. In this case, the amount of gate sharing must be limited. A more general framework for the sharing of gates across signal networks is described in [37].

4.4 Finding Enabled Cubes and Trigger Cubes

Since each region is implemented with a single cube, to obtain a hazard-free implementation, all literals in the cube must correspond to signals that are *stable*, i.e., constant throughout the excitation region. Otherwise, the single-cube cover would not cover all excitation region states. When a single-cube cover exists, an excitation region $ER(u*, k)$ can be sufficiently approximated using a cube called an *enabled cube*, denoted $EC(u*, k)$, defined on each signal v as follows:

$$EC(u*, k)(v) \equiv \begin{cases} 0 & \text{if } \forall s \in ER(u*, k)\ [val(s(v)) = 0] \\ 1 & \text{if } \forall s \in ER(u*, k)\ [val(s(v)) = 1] \\ X & \text{otherwise} \end{cases}$$

If a signal has a value of 0 or 1 in the enabled cube, the signal can be used in the cube implementing the region. A cube, such as the enabled cube, implicitly represents a set of states in the obvious way. The set of states represented by the enabled cube is always a superset of the set of excitation region states (i.e., $EC(u*, k) \supseteq ER(u*, k)$).

Each cube in the implementation is composed of *trigger signals* and *context signals*. For an excitation region, a trigger signal is a signal whose firing can cause the circuit to enter the excitation region while any non-trigger signal which is stable in the excitation region can potentially be a context signal. The set of trigger signals for an excitation region $ER(u*, k)$ can also be represented with a cube called a *trigger cube* $TC(u*, v)$ defined as follows for each signal v:

$$TC(u*, k)(v) \equiv \begin{cases} val(s'(v)) & \text{If } \exists s, s' \left[(s \xrightarrow{v} s') \wedge (s \notin ER(u*, k)) \wedge (s' \in ER(u*, k))\right] \\ X & \text{otherwise} \end{cases}$$

In order for our synthesis procedure to generate a circuit, the cover of each excitation region must contain all its trigger signals (i.e., $C(u*, k) \subseteq TC(u*, k)$). Since only stable signals can be included, a necessary condition for our algorithm to produce an implementation is that all trigger signals be stable (i.e., $EC(u*, k) \subseteq TC(u*, k)$). If a trigger signal is not stable then we must either constrain concurrency [47], add state variables [37], or use a more general algorithm [7].

The enabled cubes and trigger cubes are easily found with a single pass through the RSG. Table 1 shows the enabled cubes and trigger cubes corresponding to all the excitation regions in the SEL.

Table 1: Enabled cubes and trigger cubes for the SEL.

$u*, k$	$EC(u*, k)$	$TC(u*, k)$
$xfer_o \uparrow, 0$	11X0100X010	XXXX1XXXXXX
$xfer_o \uparrow, 1$	110X010X001	XXXXX1XXXXX
$xfer_o \downarrow, 0$	0X00XX10000	0XXXXXXXXXX
$data_o \uparrow, 0$	10000000X00	1XXXXXXXXXX
$data_o \downarrow, 0$	11X010X1010	XXXX1XXXXXX
$data_o \downarrow, 1$	110X01X1001	XXXXX1XXXXX
$sel_o \uparrow, 0$	1000000X000	1XXXXXXXXXX
$sel_o \downarrow, 0$	11100001110	XXXXXXXXX1X
$sel_o \downarrow, 1$	11010001101	XXXXXXXXXX1
$out1_o \uparrow, 0$	11100001100	X11XXXXXXXX
$out1_o \downarrow, 0$	1XX01010010	XXXXXX10XXX
$out2_o \uparrow, 0$	11010001100	X1X1XXXXXXX
$out2_o \downarrow, 0$	1X0X0110001	XXXXXX10XXX

$\langle xfer_i, data_i, sel1_i, sel2_i, out1_i, out2_i, xfer_o, data_o, sel_o, out1_o, out2_o \rangle$

4.5 Finding an Optimal Correct Cover

Our procedure to find a correct cover begins with a cube consisting only of the trigger signals (i.e., $C(u*, k) = TC(u*, k)$). If this cover contains no states that violate either the covering or entrance constraint, we are done. This, however, is often not the case, and context signals must be added to the cube to remove any violating states. For each violation detected, the procedure determines the choices of context signals which would exclude the violating state. Finding the smallest set of context signals to resolve all violations is a covering problem.

Due to the implication in the entrance constraint, inclusion of certain context signals may introduce additional violations which must be resolved. Therefore, the covering problem is *binate*.

To solve our binate covering problem, we create a *covering and closure (CC) table* [28] for each region. While other techniques exist to find binate covers such as those described in [36, 11], the CC table is simple and facilitates presentation. There is a row in the CC table for each context signal, and there is a column for each violation and each violation that could potentially arise from a context signal choice. An entry in the table contains a cross ($\times$) if the context signal resolves the conflict. An entry in the table contains a dot ($\circ$) if the inclusion of the context signal would require the violation to be resolved.

To construct the table for a given excitation region $ER(u*, k)$, the procedure first finds all states in the initial cover (i.e., $TC(u*, k)$) which violate the covering constraint. In other words, a state s in $TC(u*, k)$ is a violating state if the signal u has the same value but is not enabled (i.e., $s(u) = 0$ for a set region or $s(u) = 1$ for a reset region), is enabled in the opposite direction (i.e., $s(u) = F$ for a set region or $s(u) = R$ for a reset region), or is enabled in the same direction but the state is not in the current excitation region (i.e., $s(u) = R$ for a set region or $s(u) = F$ for a reset region and $s \notin EC(u*, k)$). If a violation exists, the procedure adds a new column to the table with a cross in each row corresponding to a context signal v that would exclude the violating state (i.e., $EC(u*, k)(v) = \neg val(s(v))$).

The next step in the table construction is to find all state transitions which violate the entrance constraint in the initial cover or may violate it for a particular context signal choice. For any state transition $s \xrightarrow{v} s'$, this is possible when s is not in the excitation region (i.e., $s \notin EC(u*, k)$), s' is a quiescent state (i.e., $s'(v) = 1$ for a set region and $s'(v) = 0$ for a reset region), s' is in the initial cover (i.e., $s' \in TC(u*, k)$), and v excludes s (i.e., $EC(u*, k)(v) = \neg val(s(v))$). For each entrance constraint violation or potential violation detected, the procedure adds a new column to the table again with a cross in each row corresponding to a context signal that would exclude the violating state. If the signal v in the state transition is a context signal, the state s' only needs to be excluded if v is included in the cover. This implication is represented with a dot being placed in the row corresponding to the signal v.

If a violation is detected for which there is no context signal to resolve it, the CC table construction fails. In this case, as with non-stable trigger signals, it is necessary to constrain concurrency, add state variables, or use a more general algorithm.

In a single pass through the RSG, all the CC tables can be constructed. When implementing $(out2_o \downarrow, 0)$ from the SEL, no covering constraint violations are detected. This is not surprising since our complex-gate implementation of this region only contained the trigger signals $xfer_o$ and $\neg data_o$. There are, however, entrance constraint violations which are shown in the CC table in Table 2.

Table 2: CC table for $(out2_o \downarrow, 0)$ from the SEL.

Signal	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$xfer_i$		×		×	×								×		
$sel1_i$	×	○	×		○	×	×	○		○		○	×		○
$out1_i$	×	×	○	○		×	×	×	×	×	○		○	×	×
$out2_i$	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
$xfer_o$															
$data_o$															
sel_o															
$out1_o$	○					×	○	×	○					×	×
$out2_o$	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

The last step is to find the smallest set of context signals to implement each excitation region by solving the CC tables that are constructed. The CC tables are solved using standard reduction rules [28] given below:

Rule 1: (Select essential rows) If a column contains only a single cross and blanks elsewhere, then the row with the cross must be selected. The row is deleted together with all columns in which it has crosses.

Rule 2: (Remove columns with only dots) If a column has only a single dot and blanks elsewhere, the row containing the dot must be deleted together with all columns in which it has dots.

Rule 3: (Remove dominating columns) A column C_j dominates a column C_i if it has all the crosses and dots of C_i. If C_j dominates C_i, then C_j is deleted.

Rule 4: (Remove dominated rows) A row R_i dominates a row R_j if it (a) has all the crosses of R_j; and (b) for every column C_p in which R_i has a dot, either R_j has a dot in C_p or there exists a column C_q in which R_j has a dot, such that, disregarding the entries in rows R_i and R_j, C_p dominates C_q. If R_i dominates R_j, then R_j is deleted together with all columns in which it has dots.

Rule 5: (Remove rows with only dots) If a row only has dots, then the row is deleted together with all columns in which it has dots.

It is important to note that when applying rule 4, two rows may mutually dominate each other. These ties are resolved by picking the rule that provides symmetry between different regions of the same signal. This symmetry often leads to gates being shared between regions. The table is completely solved when all columns are eliminated, and the context signals are those corresponding to the essential rows selected by Rule 1. While in practice these reduction rules are often sufficient to solve the table, some tables may be *cyclic*. To solve the cyclic table, we use a branch and bound method.

For the SEL, the CC table for $(out2_o \downarrow, 0)$ is reduced to the one shown in the left five columns of Table 3 after removing dominating columns. The CC table is further reduced to the single rightmost column of Table 3 after removing dominated rows. This leaves us with a choice of using either $out2_i$ or $out2_o$ as a context signal. In this case, they are equivalent, and we arbitrarily select $out2_i$.

Table 3: CC table for $(out2_o \downarrow, 0)$ from the SEL after removing dominating columns.

Signal	9	11	12	14	14
$xfer_i$					
$sel1_i$			∘		
$out1_i$	×	∘		×	
$out2_i$	×	×	×	×	×
$xfer_o$					
$data_o$					
sel_o					
$out1_o$	∘			×	
$out2_o$	×	×	×	×	×

For the SEL, we derive a gate-level timed circuit implementation with 27 literals shown in Figure 38(a). If all the timing information is ignored, we obtain a gate-level speed-independent circuit implementation with 44 literals shown in Figure 38(b). Besides being nearly 40 percent smaller, the timed circuit has reduced latency since it requires gates with at most 3-inputs while the speed-independent circuit requires many large gates including one with 6-inputs.

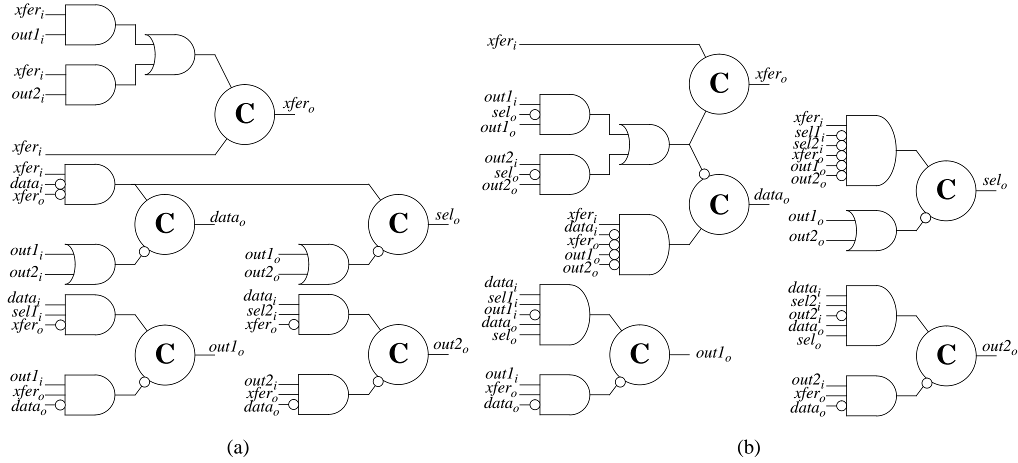


Figure 38: Gate-level (a) timed and (b) speed-independent circuits for the SEL.

4.6 Synthesis Results

Our synthesis results are tabulated in Table 4. In addition to the SEL described above, another design of the SEL (SEL2) is given in the table in which the selection of the output port is performed using a single conditional signal rather than dual-rail encoding and three signal wires. Another example in the table is a memory management unit (MMU) which was originally designed speed-independently in [50]. The last two examples, a DRAM controller (DRAM) and the target-send burst-mode portion of a SCSI controller (TSBM), were originally specified using burst-mode finite-state machines in [58].

First, we compared the literal counts (Lit) for the gate-level timed circuits derived using the generalized C-element (gC) technique and our standard C-implementation techniques. Our results show only about a 10 percent increase in literal count for generating a safe implementation that has no internal hazards. For the first three examples, the timed implementations are compared with those produced by `SYN` and `SIS` in terms of area represented by transistor count and delay represented as ratio of fanout of four inverter delays. The timed implementations are about 40 percent smaller and faster than the speed-independent ones produced by `SYN`. Compared with `SIS`, the area gains are about the same, but the improvement in delay is now about 50 percent. The table also gives the number of reachable states ($|\Phi|$) for timed and other methods showing up to two orders of magnitude less states in the timed case. In fact, due to the large state space size of the MMU example, `SIS` runs out of memory during synthesis. The last two examples are compared with the

3D method with the 3D specifications and results taken from [81] assuming a $0.3ns$ inverter delay in a $0.8\mu m$ CMOS process. For these designs, our timed circuits show about a 30 percent improvement in area (comparing literal count) and delay.

Table 4: Synthesis results.

	Timed					Other Design Methodologies						
		gC	ATACS				SYN		SIS		3D	
Ex.	$\mid\Phi\mid$	Lit	Lit	Area	Del	$\mid\Phi\mid$	Area	Del	Area	Del	Lit	Del
SEL	53	25	27	104	5	256	160	7	158	11	n/a	n/a
SEL2	36	19	21	76	5	128	108	6.5	130	11.5	n/a	n/a
MMU	187	56	62	210	4.5	23,296	412	10	out of mem		n/a	n/a
DRAM	79	38	38	110	5.5	n/a	n/a	n/a	n/a	n/a	46	7
TSBM	113	32	33	140	4.5	n/a	n/a	n/a	n/a	n/a	58	7.5

Appendix

Each of the synthesis steps described in this chapter has been implemented within `ATACS`. After generating a RSG using the commands described in the previous chapters' appendices, the next step of synthesis is to use the command *findreg* to find the excitation regions and trigger signals, represented using enabled and trigger cubes. In verbose mode, the enabled and trigger cubes are output to a file named ⟨filename⟩.es. An example of such a file for the SCSI protocol controller is shown in Figure 39. The actual regions can be displayed or printed using the command *printreg* which generates a file for each region and each set of quiescent states which can be viewed using Tom Rokicki's `parg` program. While finding the regions, if a trigger signal is found to not be stable in the excitation region, an exception is raised which tells which rules are not stable, or *persistent*. It may be possible to solve this persistency problem by adding additional rules to the specification. The command *addpers* attempts to find such rules.

```
REGIONS:
STATEVECTOR:INP go INP ack q req rdy
EVENT: ENABLED CUBE: TRIGGER CUBE
+q    : 0X010 : 0XX1X
−q    : 01101 : XXXX1
+req  : X0000 : X0XX0
−req  : 01110 : X11XX
+rdy  : 01100 : XXX0X
−rdy  : 1X001 : 1XXXX
```

Figure 39: Enabled cubes and trigger cubes from the SCSI protocol controller.

The next step of synthesis is to find the conflicts and generate the CC tables using the command *findconf*. In verbose mode, the CC tables are written to a file named ⟨filename⟩.crt. An example of a CC table from the SCSI protocol controller is shown in Figure 40. It is possible that no context signal can be found to solve a conflict, and an exception occurs. To solve this problem or a persistency problem, the command *exact* can be used to switch to a more general algorithm which is described in [7]. Essentially, this algorithm allows multiple-cubes to be used to cover a single excitation region.

After the tables are generated, they are solved using the command *findcover*. In verbose mode, a list of context signals which need to be added are stored to a file named ⟨filename⟩.cr. If the table is cyclic, an exception occurs. A heuristic routine to solve cyclic

```
CONTEXT RULE TABLES:
STATEVECTOR:INP go INP ack q req rdy
Context Rule Table for rdy+
1F00F: ~go q ~rdy
FF000: ~go q
1000F: ~go ack q ~rdy
F00R0: ~go ack q
0F000: q
000R0: ack q
```

Figure 40: CC table from the SCSI protocol controller.

tables has been implemented which arbitrarily selects one signal to add as a context signal then reattempts to solve the table. This routine is invoked using the command *resolve*.

Interfaces to other synthesis systems is also provided, and they were used to do the comparisons in this chapter. The commands *sis* and *syn* change modes between using `ATACS`, `SIS`, and `SYN` for synthesis. The command *storeg* stores a graph file which can be used as input to `SIS`. The command *storesg* stores a SG file which can be used as input to `SYN`. The command *storenet* stores a circuit net file which can be read into `SYN`.

Using the command *storeprs*, the final synthesized circuit is output as production rules [44] which are stored to a file named ⟨filename⟩.prs. The command *gatelevel* can be used to toggle between finding a generalized C-implementation and the gate-level standard C-implementation. The production rules for the SCSI protocol controller are shown in Figure 41. Each production rule consists of a type, a signal name, and a guard. The guard is a conjunction of signals which represents a cube in the implementation. If the type and signal name of the production rule are of the form $+s$, the guard represents a cube from the set network for the signal s. If they are of the form $-s$, it is from the reset network. If there is no type given, then the guard represents a combinational implementation. If they are of the form ˜s, then the guard represents an inverted combinational implementation.

```
[+q: (~go & req)]
[−q: (rdy)]
[+req: (~ack & ~rdy)]
[−req: (ack & q)]
[+rdy: (q & ~req)]
[−rdy: (go)]
```

Figure 41: Production rules from the SCSI protocol controller.

Chapter 5

Technology Mapping

Come, and take choice of all my library
—William Shakespeare

The previous chapter introduced an automatic procedure for the synthesis of gate-level timed circuits and demonstrated that timed designs can be significantly smaller and faster than designs generated using other asynchronous design methodologies. These timed designs, however, are synthesized without considering explicitly the available gate library. In particular, these designs may require gates with a large number of inputs which is not practical for existing technologies. In CMOS, for example, gates with more than four transistors in series are typically considered to be too slow, and they must be decomposed. While in a synchronous design high-fanin gates can be decomposed in an arbitrary manner, in an asynchronous design decomposition must be done in such a way as to not introduce hazards. This chapter addresses the problem of finding hazard-free mappings of timed circuits to limited-fanin gate libraries.

It has been shown for fundamental mode asynchronous circuits that synchronous technology mapping techniques can be applied with small modifications to account for hazards [65]. The fundamental-mode assumption, however, limits the concurrency that can be specified so these results cannot be applied to our timed circuit design style.

Technology mapping of speed-independent circuits has also been addressed [6, 64, 7]. The techniques employed use heuristics to investigate various decompositions, and when necessary add additional connections called *acknowledgment wire forks* to restore correctness to the decomposed implementation. These forks increase both the fanin and fanout of the gates in the implementation degrading the performance. These techniques also do not take

timing into account and would produce unnecessarily conservative and possibly incorrect timed circuit implementations.

To our knowledge, the only procedure for technology mapping of asynchronous circuits that takes timing into account is the one within Berkeley's SIS [38]. We have shown in the previous chapter that the implementations that are produced by SIS can be inefficient in terms of circuit area and delay due to the cost of the delay elements that must be added to remove hazards and the fact that timing information is neglected until late in the design process.

In this chapter, we describe an automatic procedure to map timed circuits to practical gate libraries without needing to add any delay elements. Beginning with a specification, an unlimited-fanin circuit implementation, and a gate library description, an automatic procedure is employed to investigate possible decompositions of any gates larger than those found in the gate library. Timing information is utilized to significantly reduce the size of the search space. From this reduced search space, each decomposition is employed to guide the resynthesis of a hazard-free timed circuit which is then mapped to the given gate library.

5.1 Gate Libraries

The general structure of our implementations is in the from of a standard C-implementation as depicted in Figure 37. In this structure, the upper sum-of-products represents the logic for the set, the lower sum-of-products represents the logic for the reset, and the result is merged with a C-element. When available in the gate library, this structure can be implemented directly in CMOS as a single compact generalized C-element with weak-feedback as shown in Figure 35(b) or as a fully-static generalized C-element as shown in Figure 35(c) [44]. When these complex gates are not available, the standard C-implementation structure is constructed using combinational gates and a C-element. If the library includes AND-OR-INVERT blocks, the sum-of-products may be mapped to them, otherwise discrete AND gates and OR gates must be used. We require that the given gate library contain at least 2-input AND gates, OR gates, and C-elements with arbitrary inverted inputs. The presence of AND-OR-INVERT blocks and generalized C-elements is optional.

Since delays for transitions on output signals must be specified before the gates generating them are produced, it is necessary to have a good estimate of the delay of these gates

to produce efficient timed circuits. The solution that we propose is to use a delay of 0 for the lower bound and an automatic analysis of the given library to derive the upper bound of the delay from the largest gate structure of the form shown in Figure 37 that can be built from the limited-fanin gates found in the library. Using this technique to estimate delays, however, means that when a network of gates for an output signal includes a high-fanin gate which must be decomposed to multiple levels of logic, the delay associated with transitions on this output signal may be larger than originally estimated. This increase in delay must be reflected in the specification, and it may change the resulting implementation. Since decomposition techniques for speed-independent designs do not take this into consideration, they may produce incorrect circuits when naively applied to timed circuits.

5.2 Decomposition

Given an orbital net, an arbitrary gate-level timed circuit implementation, and a gate library, the goal of technology mapping is to implement the circuit using only the limited fanin gates found in the given library optimized to some cost function such as area or delay. The technology mapping procedure first decomposes each gate in the initial implementation with a fanin higher than that found in the gate library. Next, the partitioning step trivially identifies each signal network as a cone of logic. Finally, the matching and covering step is used to bind portions of each signal network to gates found in the library to produce an efficient implementation. It was shown in [64] that for speed-independent circuits the decomposition of high-fanin OR gates from the standard C-implementation structure can be done safely in any arbitrary manner, and that the synchronous matching/covering techniques can be used with minor modifications. These results can be easily extended to our class of timed circuits. However, for the AND gates, or cubes, care must be taken when decomposing them so as not to introduce hazards. Therefore, it is the decomposition of these cubes which the remainder of this section addresses.

Our procedure to decompose each high-fanin AND gate searches for a decomposition that uses the minimum number of logic levels. This is accomplished by adding new signals to the original specification which can be used to decompose each high-fanin gate without changing the concurrency originally specified. Each decomposition results in a modified specification which is then resynthesized to obtain a new timed circuit implementation that is guaranteed to be hazard-free. This decomposition procedure first attempts to decompose

the circuit using one new signal for each high-fanin gate. If no such decomposition can be found that successfully decomposes all gates to ones found in the given library, then the specification which results in the implementation that requires the smallest fanin gates is taken as the new starting point. Using this new specification, an additional signal is added to decompose each remaining high-fanin gate. This procedure terminates either when all high-fanin gates have been successful decomposed into multi-level logic implementations, or when the minimum fanin of the best implementation and the number of gates needing to be decomposed is no longer decreasing. In the remainder of this section, we explain our decomposition procedure in more detail.

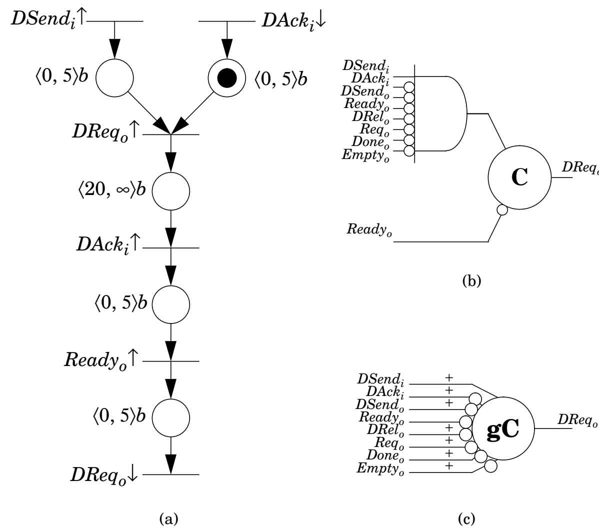


Figure 42: (a) Part of the orbital net for the *tsbm*, (b) a standard C-implementation, and (c) a generalized C-implementation of the signal $DReq_o$.

5.2.1 Searching the Decomposition Space

A decomposition of a cube is a partition of the set of trigger and context signals into two subsets: an *extracted set* and a *reduced set*. The signals in the extracted set are used as trigger signals for a transition on a new signal that is added to decompose the high-fanin gate. For a particular cube composed of n signals, there are $2^n - 1$ different decompositions. For example, the 8-input AND gate in Figure 42(b) which must be decomposed has 255 different decompositions.

Fortunately, we do not need to examine all of them as many decompositions which never

lead to a successful decomposition can be safely eliminated from consideration. When two signal transitions are ordered, if the signal with the later transition is extracted as a trigger signal for the new signal transition, the signal with the earlier transition need not also be extracted. The earlier transition, if extracted, would not be a trigger signal for the new transition as two trigger signals are *never* ordered. If the signal with the earlier transition is needed in the implementation of the new signal transition, it is as a context signal.

We use the above intuition in two ways. First, since all trigger signal transitions occur later than any context signal transition, any decomposition with an extracted set that contains both trigger and context signals from the original gate is eliminated. Second, a timing analysis algorithm such as the one described for deterministic specifications in Chapter 3 or for more general specifications in [35] is used to determine the order of context signal transitions. Any decomposition composed of two context signals that have ordered transitions is eliminated. By taking advantage of ordering information, the number of possible decompositions for the 8-input AND gate from the *tsbm* is reduced from 255 to only 23.

5.2.2 Decomposition Through Resynthesis

For each signal which needs to be decomposed, our procedure selects a decomposition from the set of potential decompositions that remains after applying the criterion described in the previous subsection. The original specification is then modified to incorporate a new signal for each signal being decomposed. For simplicity, we explain here the case in which the orbital net does not contain conditional behavior, or choice. We describe an example with choice later.

The procedure first adds a rising transition for each new signal to the orbital net. For each signal in the extracted set, this new transition has a behavior place in its preset from the corresponding transition. The timing requirements on these places have a lower bound of 0 with an upper bound derived as mentioned earlier from the maximum delay for a limited-fanin standard C-implementation. If the extracted set is composed of trigger signals, the original connections (places and transitions) between the corresponding transitions on these trigger signals and the rising (falling) transition on the signal being decomposed are replaced by a single behavior place which is added to the postset of the new rising transition. If the extracted set is composed of context signals, a constraint place with timing requirement $\langle 0, \infty \rangle c$ is added to the postset of the new rising transition and the preset of the original

rising (falling) signal transition. When the falling (rising) transition of a signal also needs to be decomposed, it is done with the falling transition of the new signal using the same procedure just described. Otherwise, the falling transition of the new signal is placed between all the trigger signals for the original falling (rising) transition and the original falling (rising) transition itself.

This new specification is then resynthesized using the automatic procedure from [54] to produce a new hazard-free timed circuit implementation. If the new implementation does not have any high-fanin gates, the decomposition is successful. Otherwise, the procedure must repeat using a different decomposition for each remaining high-fanin gate.

Returning to the *tsbm*, we apply our technology mapping procedure to the specification and implementation shown in Figure 42 with a gate library that contains 4-input AND gates, OR gates, C-elements, and generalized C-elements. One decomposition for the 8-input AND gate from the *tsbm* has an extracted set that contains only the trigger signal $DSend_i$. The portion of the new orbital net corresponding to this decomposition is as shown in Figure 43(a). This new specification results in the generalized C-implementation shown in Figure 43(b). Unfortunately, this decomposition results in an implementation that requires a 7-input gate. Another possible decomposition is the one with an extracted set that contains just the context signal $\neg DSend_o$ which results in the portion of the orbital net shown in Figure 44(a). Note that the place between the new signal transition $x_1 \uparrow$ and the transition on the signal being decomposed $DReq_o \uparrow$ is now a constraint place. This decomposition produces an implementation which requires only one 2-input gate (note the generalized C-element for x_1 only requires at most two transistors in series) and one 3-input gate shown in Figure 44(b).

Various cost functions can be used to evaluate different successful decompositions in terms of circuit area and delay. Because the number of different decompositions is usually small, it may be computationally feasible for the decomposition procedure to analyze each decomposition, and select the one with the lowest cost that decomposes all high-fanin gates to gates found in the library. As a heuristic to speedup the process, our procedure exits after a decomposition is found that decomposes each high-fanin gate to the limited-fanin gates in the given library. Although a better decomposition may exist, due to a good ordering heuristic employed, the first successful decomposition found is typically close to the optimal in terms of area and delay.

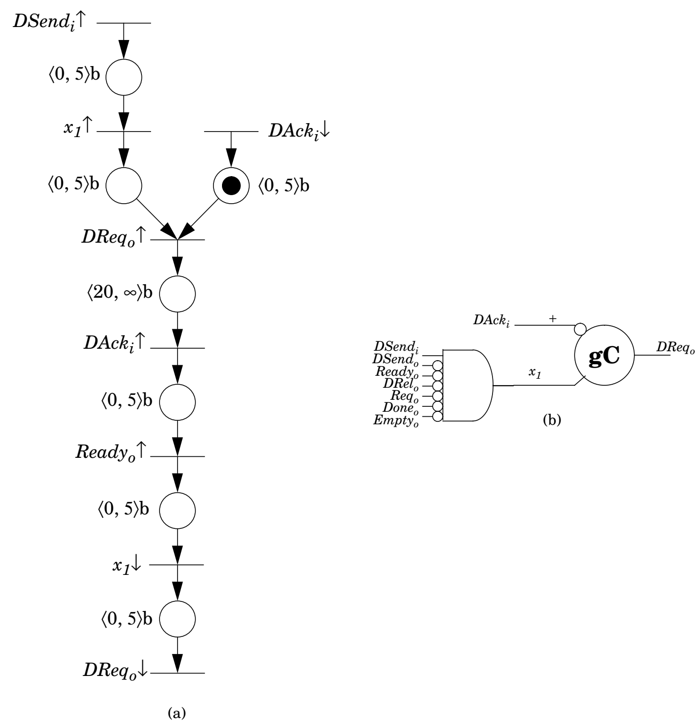


Figure 43: (a) Part of the orbital net for a decomposition using a trigger signal, and (b) corresponding generalized C-implementation.

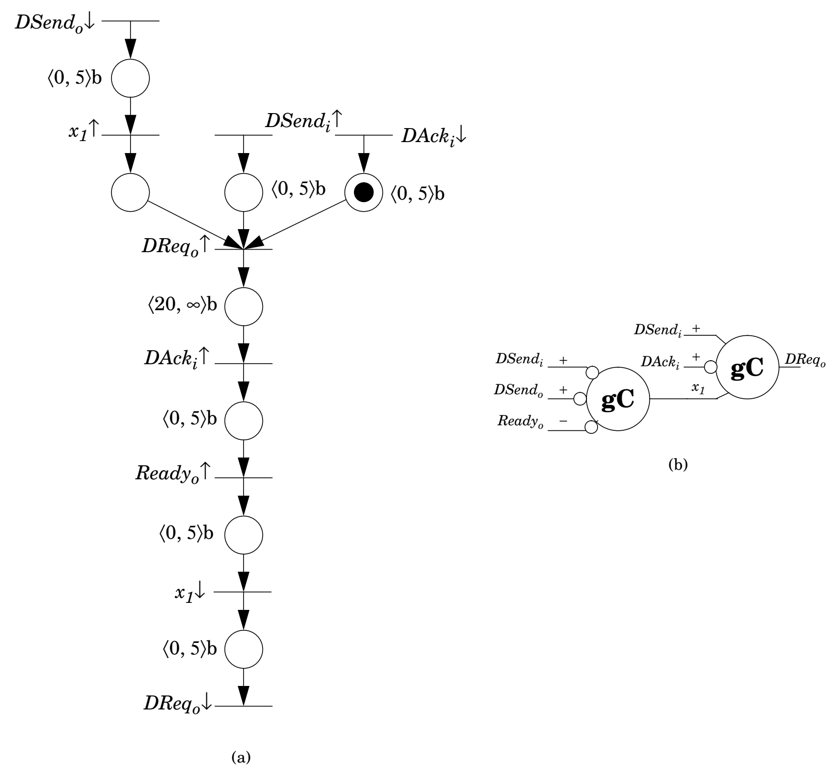


Figure 44: (a) Part of the orbital net for a decomposition using a context signal, and (b) corresponding generalized C-implementation.

5.2.3 Multi-level Decompositions

If the procedure is not successful at decomposing all high-fanin gates by adding only one additional signal, the decomposition procedure is iterated to produce multiple levels of logic. After the first pass, if all gates have not been successfully decomposed, the procedure selects the decomposition for each gate which requires the minimum gate size and uses its corresponding specification and implementation as input to a new iteration of the procedure. This step is repeated until an implementation is returned that either uses only gates in the library or has a minimum gate size that is no longer decreasing. In the second case, our procedure is unable to generate an implementation using the given specification and gate library. To handle this situation, either the requirements in the specification must be relaxed or carefully designed atomic gates may need to be added to the gate library.

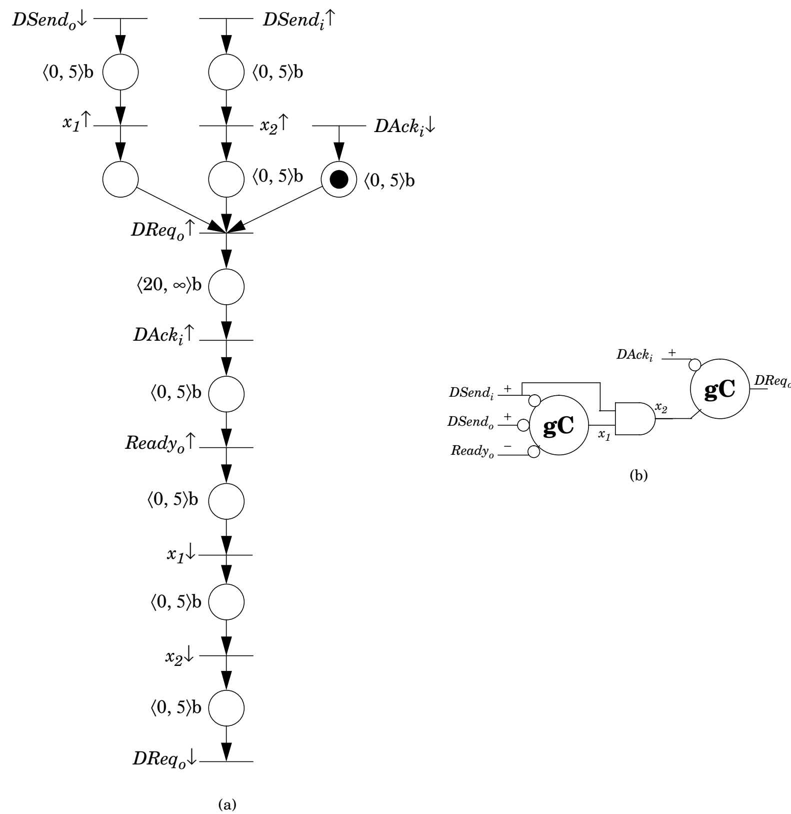


Figure 45: (a) Part of the orbital net for a multi-level decomposition, and (b) corresponding generalized C-element implementation of $DReq_o$ with a maximum fanin of two.

For the *tsbm* example, if we reduce the library size to include only 2-input gates, it can no longer be decomposed using only one new signal. The best decomposition that the procedure finds is the one shown in Figure 44(b) which uses only one 3-input gate which must be further decomposed. The orbital net shown in Figure 44(a) is now taken as the initial specification and the circuit shown in Figure 44(b) is the initial implementation. For this new iteration, the procedure adds an additional signal x_2 to decompose the 3-input gate. A portion of the orbital net for a decomposition that extracts the trigger signal $DSend_i$ is shown in Figure 45(a). Synthesis applied to this net results in a generalized C-implementation shown in Figure 45(b) using three 2-input gates. Note that x_1 is a context signal in the implementation shown in Figure 44(b), and for that reason, it can move to the gate implementing x_2.

5.3 Example

We present another example, an optimized version of the port selector (*SELopt*), to illustrate the application of our decomposition procedure to a circuit with conditional behavior, or choice. Part of the original orbital net for the *SELopt* is shown in Figure 46(a), and the original gate-level timed circuit implementation is shown in Figure 47(a). If we restrict our library to gates with a maximum fanin of 3, there is a 4-input AND gate that is shared to implement sel_o and $data_o$ which must be decomposed. A new signal is added for each of these signals, but we concentrate on the signal x_1 which is added to decompose sel_o. Part of the orbital net after a decomposition of sel_o is shown in Figure 46(b). This decomposition has an extracted set which consists of the context signals $\neg out1_o$ and $\neg out2_o$. The procedure detects that the corresponding transitions on these context signals occur on disjoint paths, so these transitions share a single place that is added to the preset of $x_1 \uparrow$. Since there are two falling transitions on the signal sel_o, the procedure adds two falling transitions on the new signal x_1. Applying synthesis to this new specification produces the decomposed implementation shown in Figure 47(b). If we further restrict the library to only contain 2-input gates, there are three gates which must be decomposed. The resulting implementation is shown in Figure 47(c).

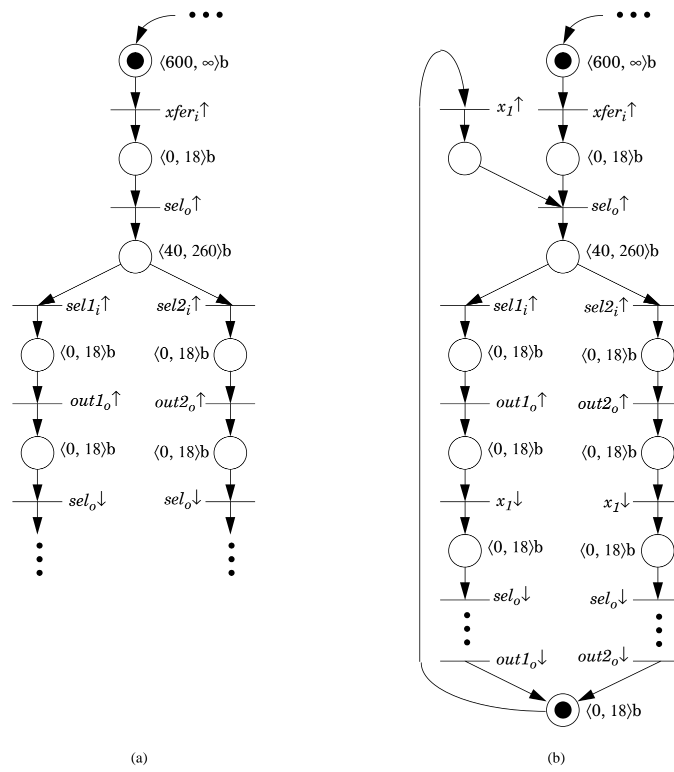


Figure 46: (a) Part of the orbital net for the *SELopt*, and (b) part of the orbital net after a decomposition of the signal sel_o.

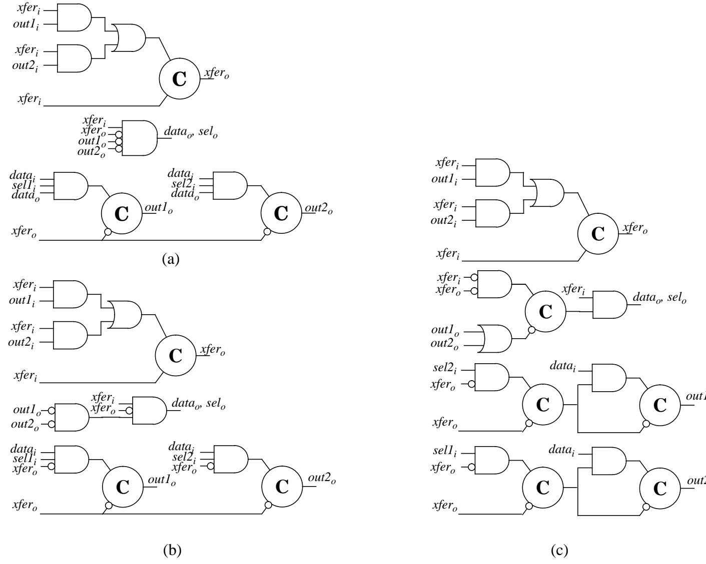


Figure 47: The gate-level timed circuit implementation of the *SELopt* (a) before decomposition; after decomposition to (b) 3-input gates and (c) 2-input gates.

5.4 Technology Mapping Results

The decomposition procedure has been used to map several examples as reported in Table 5. First, a timed version of the target-send burst-mode (*tsbm*) cycle of a SCSI data transfer controller [82] is synthesized using gate libraries with a maximum fanin of 4, 3, and 2. The next three rows are implementations of the optimized port selector (*SELopt*) [54] also using libraries with a fanin of 4, 3, and 2. The last example is an asynchronous memory management unit [50].

The gate library used for each example contains gates with a maximum fanin size as specified in parentheses next to the name of the example. The next two columns give the number of gates in the standard C-implementation as well as the number of gates that are larger than the maximum fanin and must be decomposed. All high-fanin gates were successfully decomposed. The area and latency for the decomposed standard C-implementation are given in the next two columns followed by the area and latency after the implementation is mapped to a library which contains generalized C-elements. Area is reported as the implementation's transistor count. Latency is the longest delay through a block of logic generating an output transition driving a fanout of 4, and it is reported normalized to the delay of a single inverter (about $300ps$ for $0.8\mu m$ CMOS process at nominal conditions). Here, we

Table 5: Technology mapping results.

Example	# of Gates	# of Gates to Decompose	AND/OR/C Area (xtors)	AND/OR/C Latency (inv)	gC Library Area (xtors)	gC Library Latency (inv)	Iter (num)
tsbm (4)	15	1	122	8.1	70	4.9	1
tsbm (3)	15	1	122	8.1	70	4.9	1
tsbm (2)	15	3	154	7.2	87	5.7	25
SELopt (4)	11	0	66	5.3	45	3.8	0
SELopt (3)	11	2	70	5.3	53	3.8	1
SELopt (2)	11	4	108	8.5	67	4.2	3
MMU (4)	27	4	186	5.8	132	5.2	14

see that being able to map the implementations to generalized C-elements produces more than a 30 percent improvement in both area and delay. Finally, the number of iterations necessary to decompose the high-fanin gates is shown. In all case, the decompositions are completed in a reasonable number of iterations.

Appendix

The decomposition procedure has been automated within ATACS. After synthesis, the resulting implementation is checked to see if it requires high-fanin gates. The maximum allowed gate size is set by the command *maxsize*. When high-fanin gates are detected, the command *breakup* can be used to run the decomposition procedure. The upper bound of the timing constraint which is added for new rules on the new signals added is set using the command *gatedelay*. The search of the decomposition space can be done both automatically or manually, set by the command *manual*. When the decomposition procedure is done manually, it prompts the user to input the decomposition to try for each gate being decomposed. The value of the decomposition determines which signals are to be in the extracted set. For a production rule $[*s : (a_0 \& a_1 \& \ldots a_n)]$, the literal a_i is included in the extracted set if the value of the decomposition AND'ed with 2^i is true. For each decomposition, the procedure first generates a new timed ER structure which describes the new orbital net and stores it to the file named ⟨filename⟩BRK.er. Synthesis is applied to this new structure, and if the circuit no longer has high-fanin gates, the procedure terminates. Otherwise, the procedure attempts a new decomposition. If no decomposition is found that breakups all gates with one new signal for each high-fanin gate, then the procedure takes the best decomposition it found so far and attempts to decompose it by adding another new signal. This procedure repeats until a decomposition is found or the maximum gate size is no longer decreasing.

Chapter 6

Design Examples

Example is always more efficacious than precept.
—Samuel Johnson

This chapter describes three examples in detail. The first is a controller for a fully asynchronous memory management unit (MMU) which is used to illustrate that significant improvements in circuit complexity can be achieved using timing constraints over traditional speed-independent design methods. The second design is an asynchronous DRAM controller which interfaces with a synchronous environment. The resulting implementation is compared with a synchronous implementation designed using the synchronous synthesis package within Berkeley's `SIS`. Considering the clock as just another input, synchronous circuits can also be designed using our methodology. The last example is therefore a synchronous design: a two-bit counter.

6.1 MMU Controller

The MMU is designed for use with a 16-bit asynchronous microprocessor [45], and the original implementation was derived using Martin's synthesis method [50]. The basic operation of the MMU is to convert a 16-bit *memory address* to a 24-bit *real address*. There are six possible cycles that the MMU controller can enter, depending on data from the environment. The 6 cycles can be independently designed and merged together to get the overall implementation. For simplicity of presentation, the first subsection describes only the design of one cycle: the *memory data load* (MDl) cycle. The next subsection presents the complete implementation of the MMU controller.

6.1.1 The Memory Data Load Cycle

A simplified block diagram is shown in Figure 48 in which only signals involved in the MDl cycle are depicted.

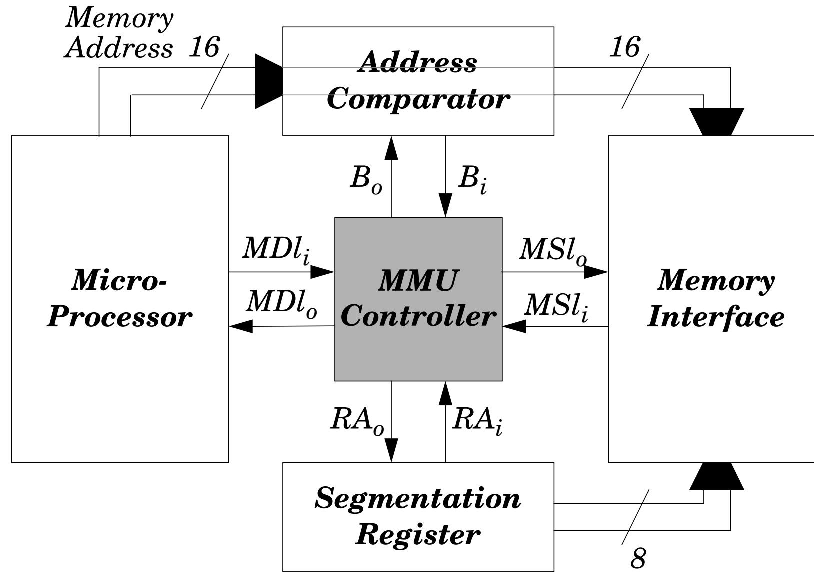


Figure 48: Block diagram for the MDl cycle of the MMU controller.

The high-level CSP specification for the memory data load cycle is:

$$*[\overline{MDl} \rightarrow (RA \parallel B); MSl; MDl]$$

This specification is initially transformed into the following *handshaking expansion*:

$$*[[MDl_i \wedge \neg RA_i]; RA_o\uparrow; [\neg B_i]; B_o\uparrow; [RA_i \wedge B_i \wedge \neg MSl_i]; MSl_o\uparrow; [MSl_i]; MDl_o\uparrow; RA_o\downarrow; B_o\downarrow; [\neg MDl_i]; MSl_o\downarrow; MDl_o\downarrow],$$

which can be converted to the constraint graph shown in Figure 49.

The transformation from a CSP specification to a handshaking expansion is not unique. A more concurrent constraint graph shown in Figure 50 also satisfies the high-level CSP specification. This specification is simply a *reshuffling* [44] of the earlier one. This reshuffling is not considered in [50] because it results in a complete state coding violation [17]. This means that the more concurrent specification cannot be implemented without adding state variables. Adding state variables not only changes the specification, but can also add extra circuitry and/or delay to the implementation. This cost often outweights the benefit

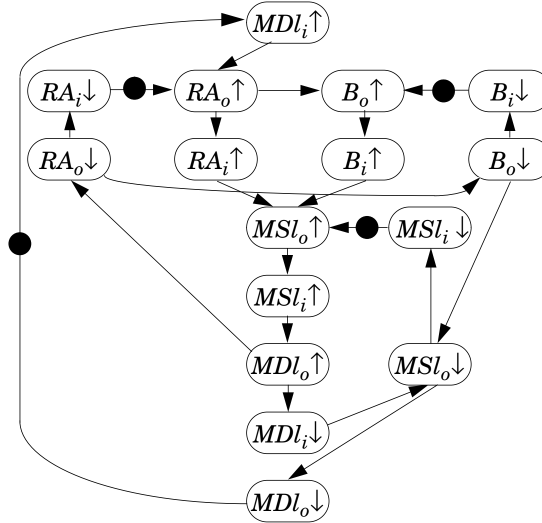


Figure 49: The cyclic constraint graph for the unoptimized MDl cycle.

of the higher degree of concurrency. This particular problem can also be solved by adding persistence rules, but this can reduce the concurrency in the specification. If conservative timing constraints are also added, the reduced state graph of the more concurrent specification shown in Figure 50 does not have a complete state coding violation, and thus, it can be implemented without adding state variables or persistence rules. To make the specification in Figure 50 persistent, three arcs are added to the constraint graph as shown in Figure 51; the specification can now be implemented speed-independently. As shown later, the speed-independent implementation is still more complex than the original implementation derived from the specification in Figure 49.

A speed-independent and a timed implementation of the specification shown in Figure 51 are compared. For the timed implementation, the timing constraints used are depicted in Figure 51. The lower bound of the timing constraint on $MDl_i \uparrow$ states that the processor does not issue memory requests faster than every 30ns. The lower bound of the timing constraint on $MSl_i \uparrow$ states that the DRAM access time takes at least 30ns. Both of their upper bounds are infinite since the processor could choose never to do a load, or the interface could choose never to process the request. The reseting of the acknowledgement (i.e., $MDl_i \downarrow$ and $MSl_i \downarrow$) is assumed to be somewhat faster, and must occur within 5 to 30ns of the reset of the request. The other numbers were obtained from SPICE simulations of the datapath

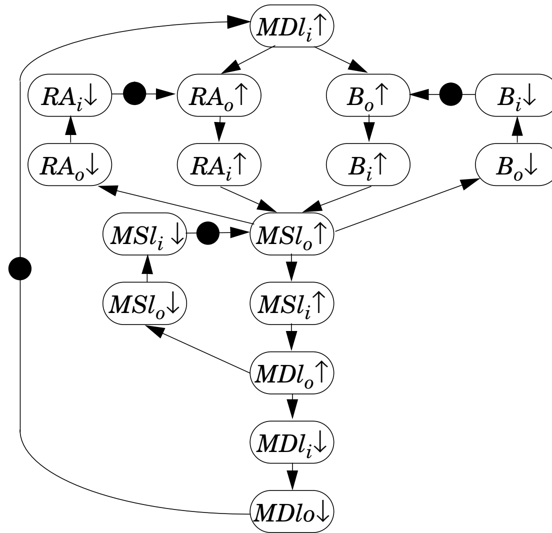


Figure 50: The cyclic constraint graph for the optimized MDl cycle.

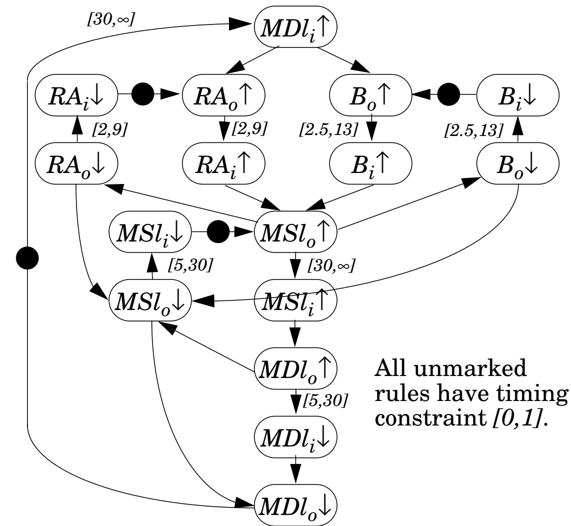


Figure 51: The cyclic constraint graph for the persistent MDl cycle.

circuitry for a $0.8\mu m$ CMOS process. The comparator, denoted B_i, has a delay of between 2.5 to 13ns, and the registers, denoted RA_i, have a delay of between 2 to 9ns depending on temperature, voltage, and processing variations. All output signals have a delay of 0 to 1ns where 1ns was found to be the maximum delay of the gates in the library used.

In the MMU specification, there are five events with multiple rules enabling them: $RA_o\uparrow$, $B_o\uparrow$, $MSl_o\uparrow$, $MSl_o\downarrow$, and $MDl_o\downarrow$. Timing analysis determines that at least one rule associated with each event is redundant. In all, 6 of the 15 rules on output signals in the original specification are redundant. This includes the 3 persistence rules. To determine which context signals must be added, the first step is to determine the reduced state graph and the enabled cube for each signal using the timing constraints. A state graph generated without any timing constraints results in 92 states while the reduced state graph only has 22 states. Using the reduced state graph, the timed implementation needs 5 context signals as opposed to 7 needed for the speed-independent implementation.

After adding context signals to our original specification, a speed-independent implementation requires 22 literals (note that we define a literal to be a signal in a guard) as shown in Table 6. The timing constraints reduce the circuit to only 10 literals. Thus, our circuit complexity is reduced by over 50 percent using conservative timing constraints. A generalized C-implementation for both is shown in Figure 52. Note that this reduction is possible not only because of removing redundant literals, but also because the gate needed for implementing RA_o and B_o can now be shared after the optimizations.

Speed-Independent PRs			**Simplified Timed PRs**		
$MSl_o \wedge MSl_i$	$\mapsto$	$MDl_o\uparrow$	$MDl_i \wedge MSl_i$	$\mapsto$	$MDl_o\uparrow$
$\neg MSl_o \wedge \neg MDl_i$	$\mapsto$	$MDl_o\downarrow$	$\neg MDl_i$	$\mapsto$	$MDl_o\downarrow$
$\neg MDl_o \wedge \neg MSl_o \wedge \neg RA_i \wedge MDl_i$	$\mapsto$	$RA_o\uparrow$	$\neg MDl_o \wedge \neg MSl_o \wedge MDl_i$	$\mapsto$	$RA_o\uparrow, B_o\uparrow$
MSl_o	$\mapsto$	$RA_o\downarrow$	MSl_o	$\mapsto$	$RA_o\downarrow, B_o\downarrow$
$\neg MDl_o \wedge \neg MSl_o \wedge \neg B_i \wedge MDl_i$	$\mapsto$	$B_o\uparrow$			
MSl_o	$\mapsto$	$B_o\downarrow$			
$RA_o \wedge B_o \wedge \neg MSl_i \wedge RA_i \wedge B_i$	$\mapsto$	$MSl_o\uparrow$	$RA_i \wedge B_i$	$\mapsto$	$MSl_o\uparrow$
$\neg RA_o \wedge \neg B_o \wedge MDl_o$	$\mapsto$	$MSl_o\downarrow$	MDl_o	$\mapsto$	$MSl_o\downarrow$

Table 6: Production rules for speed-independent and timed circuits for the MDl cycle.

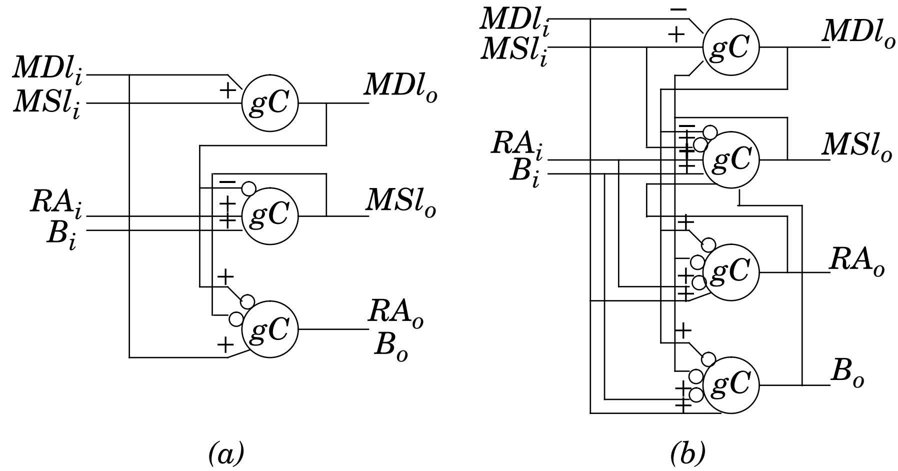


Figure 52: (a) Timed and (b) speed-independent implementations for the MDl cycle.

6.1.2 Complete MMU

The specification for the complete MMU controller process is shown in Figure 53. From this specification, our synthesis procedure obtains a reduced state graph which contains 187 states. From the reduced state graph, the procedure obtains a gate-level timed circuit implementation with 62 literals depicted in Figure 54(a) using only basic gates with at most 3-inputs. For a gate-level speed-independent circuit implementation, the state graph explodes to 23,296 states resulting in the circuit implementation shown in Figure 54(b) that is not only significantly larger, 114 literals, but also significantly slower since it requires gates as large as 8 inputs!

module *MMU*;
process *control*;
$* [[\ [MDl_i \uparrow] \rightarrow ((([RA_i \downarrow]; RA_o \uparrow) \,||\, ([B1_i \downarrow \vee B2_i \downarrow \vee B3_i \downarrow]; B_o \uparrow)); [RA_i \uparrow];$
$[\ [B1_i \uparrow \wedge LSR_i \downarrow]; \rightarrow LSR_o \uparrow; (RA_o \downarrow \,||\, B_o \downarrow); [LSR_i \uparrow]; MDl_o \uparrow; LSR_o \downarrow; [MDl_i \downarrow]; MDl_o \downarrow$
$|\ [B2_i \uparrow \wedge LSW_i \downarrow]; \rightarrow LSW_o \uparrow; (RA_o \downarrow; \,||\, B_o \downarrow); [LSW_i \uparrow]; MDl_o \uparrow; LSW_o \downarrow; [MDl_i \downarrow]; MDl_o \downarrow$
$|\ [B3_i \uparrow \wedge MSl_i \downarrow]; \rightarrow MSl_o \uparrow; (RA_o \downarrow; \,||\, B_o \downarrow); [MSl_i \uparrow]; MDl_o \uparrow; MSl_o \downarrow; [MDl_i \downarrow]; MDl_o \downarrow$
]
$|\ [MDs_i \uparrow] \rightarrow ((([WA_i \downarrow]; WA_o \uparrow) \,||\, ([B1_i \downarrow \vee B2_i \downarrow \vee B3_i \downarrow]; B_o \uparrow)); [WA_i \uparrow];$
$[\ [B1_i \uparrow \wedge SSR_i \downarrow]; \rightarrow SSR_o \uparrow; (WA_o \downarrow \,||\, B_o \downarrow); [SSR_i \uparrow]; MDs_o \uparrow; SSR_o \downarrow; [MDs_i \downarrow]; MDs_o \downarrow$
$|\ [B2_i \uparrow \wedge SSW_i \downarrow]; \rightarrow SSW_o \uparrow; (WA_o \downarrow; \,||\, B_o \downarrow); [SSW_i \uparrow]; MDs_o \uparrow; SSW_o \downarrow; [MDs_i \downarrow]; MDs_o \downarrow$
$|\ [B3_i \uparrow \wedge MSs_i \downarrow]; \rightarrow MSs_o \uparrow; (WA_o \downarrow; \,||\, B_o \downarrow); [MSs_i \uparrow]; MDs_o \uparrow; MSs_o \downarrow; [MDs_i \downarrow]; MDs_o \downarrow$
]
]]
endprocess
etc.
endmodule

Figure 53: Part of the timed HSE specification for the complete MMU controller.

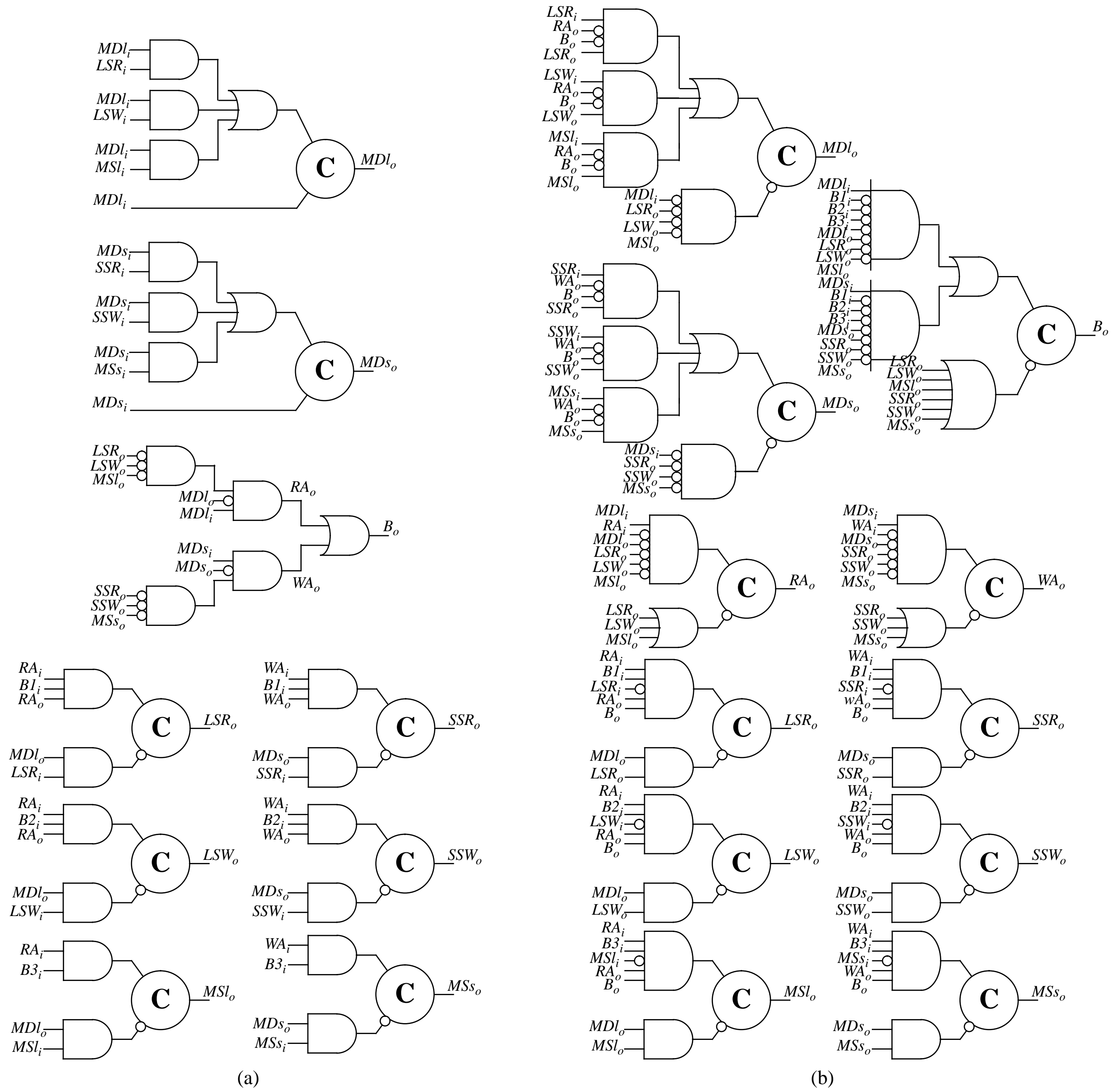


Figure 54: Gate-level (a) timed and (b) speed-independent circuits for the MMU controller.

6.2 DRAM Controller

The DRAM controller is an interface between a synchronous microprocessor and a DRAM array. Typically, a DRAM controller is implemented as a synchronous circuit. Since a DRAM controller must interface with a synchronous environment, it cannot be implemented as a speed-independent asynchronous circuit, but it can be implemented as a timed circuit that satisfies certain timing constraints.

A block diagram for the entire DRAM interface is shown in Figure 55. The DRAM controller was originally specified using the burst-mode finite-state machine representation shown in Figure 56 [58]. From the burst-mode specification, we obtained the timed HSE specification shown in Figure 57. The DRAM controller has three possible modes of operation: *refresh*, *write*, and *read*. The generalized C-implementation for the DRAM controller is shown in Figure 58. Note that while some gates are shown as multi-level implementations, they are actually implemented with single complex gates such as the one for *cas* shown in Figure 59. This implementation is not hazard-free at the gate-level. The gate-level synthesis procedure produces the gate-level hazard-free timed circuit shown in Figure 60(a). While the two implementations have a different structure, they are equivalent in terms of literal count (38 literals each) before optimizations, so there is little cost, if any, in achieving hazard-freedom at the gate-level. A synchronous implementation of the DRAM controller shown in Figure 60(b) is generated using Berkeley's synchronous synthesis program `SIS` [62]. Surprisingly, our timed design is about 40 percent smaller and 30 percent faster. This result comes from the sequential don't-care information that is taken into account by the asynchronous nature of our synthesis procedure. There is also a significant improvement in power consumption since our timed design produces no spurious transitions.

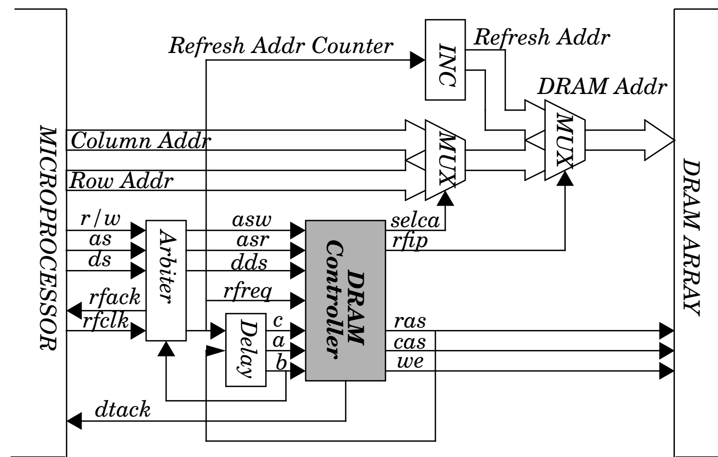


Figure 55: Block diagram for a DRAM interface.

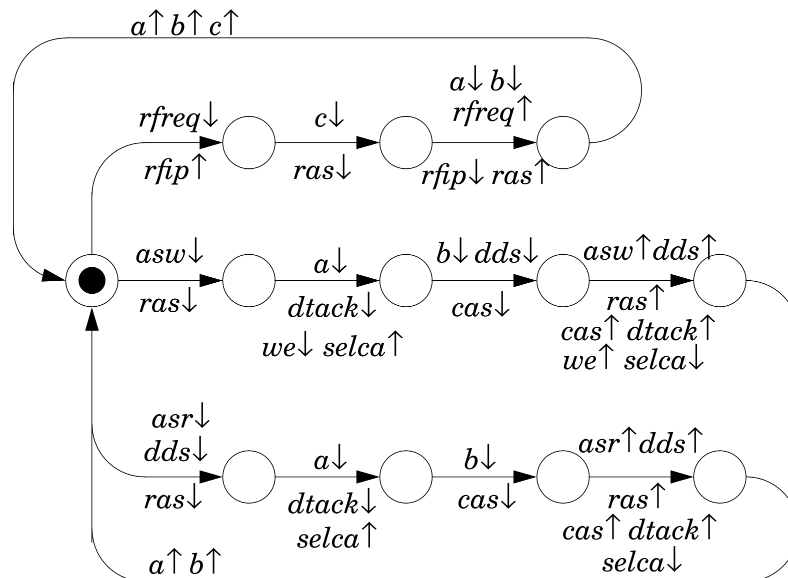


Figure 56: The burst-mode specification for the DRAM controller.

module *DRAM*;
process *control*;
$*[[\ [rfreq\downarrow] \rightarrow rfip\uparrow; [c\downarrow]; ras\downarrow; [a\downarrow \wedge\ b\downarrow \wedge\ rfreq\uparrow]; (rfip\downarrow\ \ ||\ ras\uparrow); [a\uparrow \wedge\ b\uparrow \wedge\ c\uparrow]$
$\ |\ \ [asw\downarrow] \rightarrow ras\downarrow; [a\downarrow]; (dtack\downarrow\ \ ||\ we\downarrow\ \ ||\ selca\uparrow); [b\downarrow \wedge\ dds\downarrow]; cas\downarrow;$
$\qquad [asw\uparrow \wedge\ dds\uparrow]; (ras\uparrow\ \ ||\ cas\uparrow\ \ ||\ dtack\uparrow\ \ ||\ we\uparrow\ \ ||\ selca\downarrow); [a\uparrow \wedge\ b\uparrow]$
$\ |\ \ [asr\downarrow \wedge\ dds\downarrow] \rightarrow ras\downarrow; [a\downarrow]; (dtack\downarrow\ \ ||\ selca\uparrow); [b\downarrow]; cas\downarrow;$
$\qquad [asw\uparrow \wedge\ dds\uparrow]; (ras\uparrow\ \ ||\ cas\uparrow\ \ ||\ dtack\uparrow\ \ ||\ selca\downarrow); [a\uparrow \wedge\ b\uparrow]$
$\ ]]$
endprocess
etc.
endmodule

Figure 57: Part of the timed HSE specification of the DRAM controller.

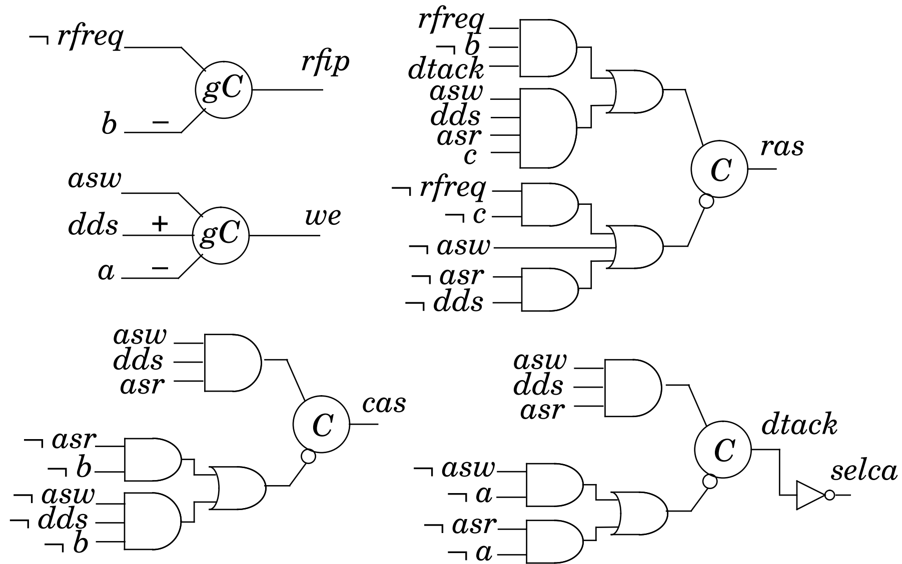


Figure 58: Overall implementation of the DRAM controller.

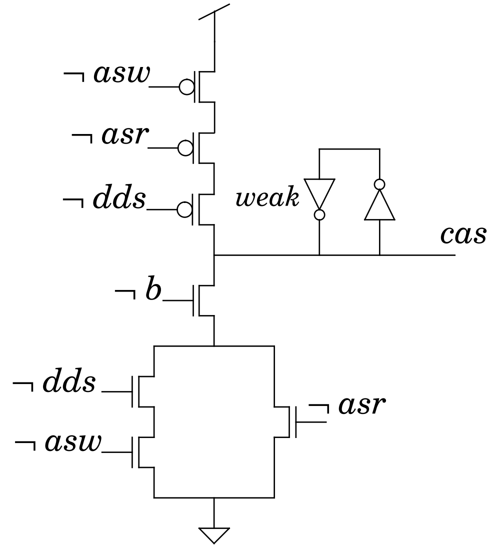


Figure 59: Complex-gate implementation of the *cas* signal for the DRAM controller.

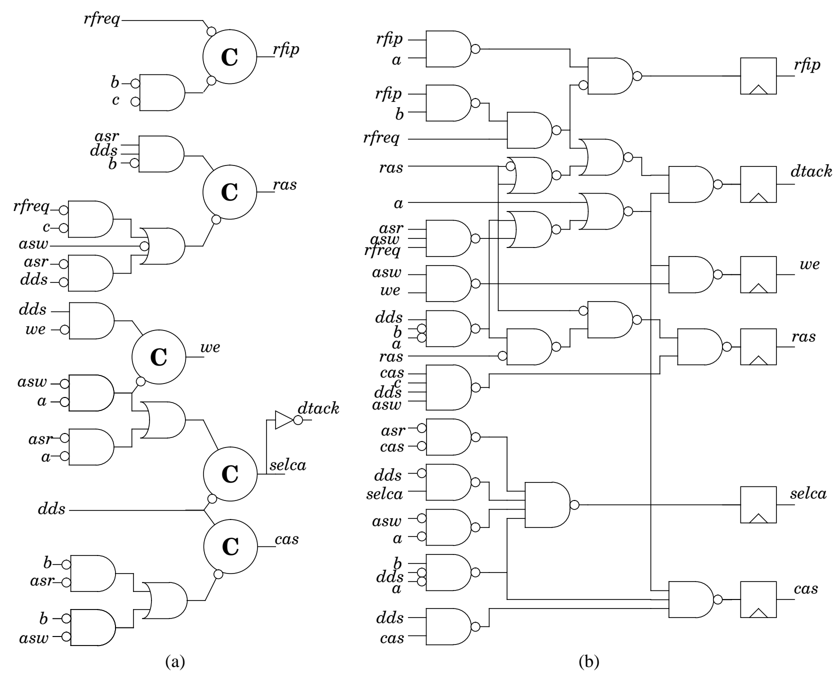


Figure 60: (a) Timed and (b) synchronous circuits for a DRAM controller.

6.3 Two-bit Synchronous Counter

The two-bit synchronous counter is specified in Figure 61(a). Additional constraints, analogous to setup times, are added to make the cyclic constraint graph strongly connected as shown in Figure 61(b). The complex gate implementation synthesized for the counter is shown in Figure 62(a). Upon closer inspection of the transistor-level diagram for this gate shown in Figure 62(b), we observe that the gates are actually typical synchronous latches, and the circuit can be redrawn as shown in Figure 62(c). This final implementation takes 6 transistors for the logic and 16 for the latches. The critical path through the logic is an inverter, a pass gate, and a latch (approximately 2.5 inverter delays). Using `SIS` and a standard synchronous gate library, the implementation for the counter shown in Figure 63 is derived. This implementation uses 32 transistors and has a critical path through an inverter and 2 NAND gates and a latch (approximately 6 inverter delays).

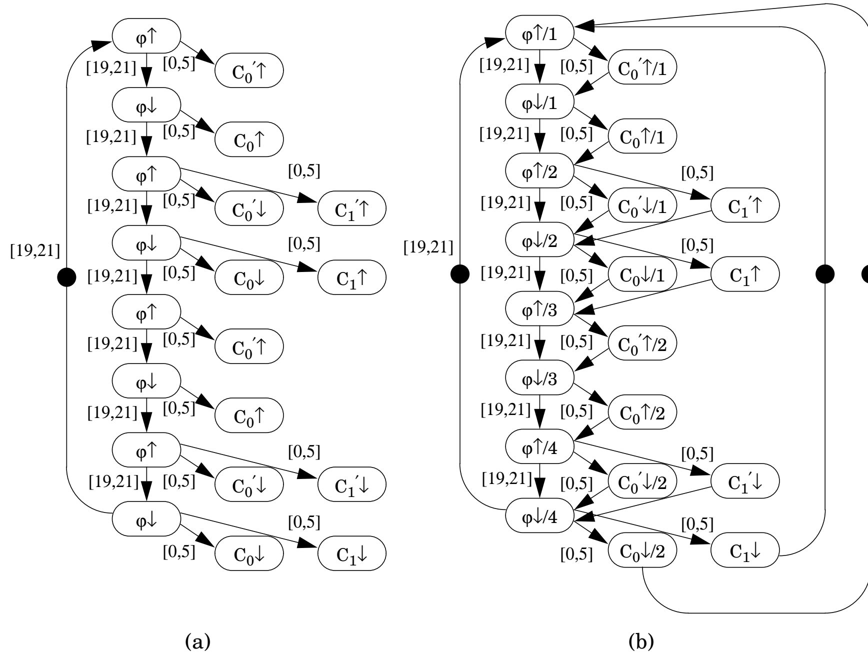


Figure 61: The cyclic constraint graph specification for a two-bit synchronous counter: (a) initial specification and (b) final specification.

Our implementation is more than 30 percent smaller and more than twice as fast as the one produced using the synchronous synthesis tool `SIS`. Comparing the implementations, we find that both implement C_0' using a single inverter. The difference is in the implementation

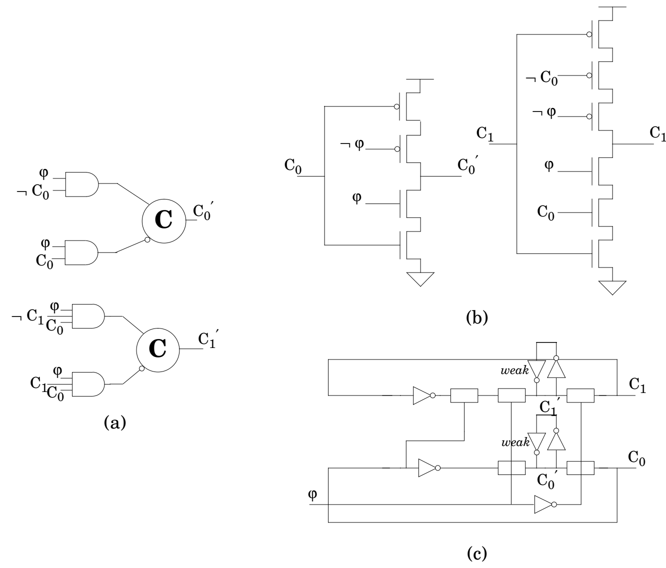


Figure 62: Complex-gate implementation of a two-bit synchronous counter.

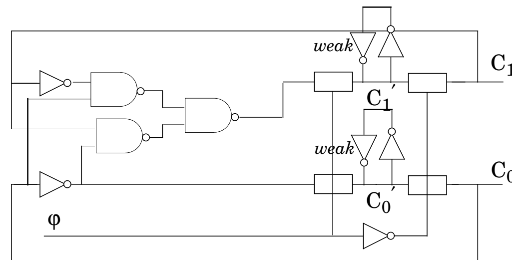


Figure 63: Implementation of a two-bit synchronous counter derived using `SIS`.

of C_1'. Our timed circuit implementation makes use of the information that C_1' only changes in states where C_0 is high. Thus, it is implemented using an inverter and a pass gate which is gated on C_0. `SIS`'s implementation, on the other hand, does not take into account the sequencing of the states. For example, if a sequence of states in which the counter is counting 00-11-01-10 were possible, this circuit would generate the correct next state given the current state. This extra logic, however, is unnecessary since this counter always goes through the states in the same order: 00-01-10-11-00,etc.

Chapter 7

Verification

Prove all things; hold fast that which is good.
—Bible

Verification is the process of checking if the circuit built satisfies its specification. There are many reasons to use verification. First, even if a circuit is automatically synthesized using a formal, systematic synthesis procedure, such as ours, verification provides a double-check to discover bugs in the synthesis tools. Second, since timing assumptions must be made at the outset to synthesize a timed circuit, verification can be used to check these assumptions after the circuit has been synthesized. Third, designers often perform hand-optimizations to synthesized circuits, and these optimizations can be checked using verification. Finally, verification can be used to measure the robustness of a design to changes in design parameters. Although a circuit may be synthesized for one set of bounded delays, it may still work when some of the delay bounds change.

Our verification procedure requires both a specification and circuit implementation either given in or translated to an orbital net representation. The orbital net for the specification is *mirrored* (i.e., inputs and outputs are swapped) [24] and composed with the orbital net for the implementation. The state space is then explored using the partial order timing analysis algorithm described earlier. If in the process of exploring the state space a failure is detected, an error trace is returned, otherwise the timed circuit is found to implement its timed specification.

7.1 Behavioral Semantics

In order to verify our timed circuits, we adopt as our behavioral semantics trace theory as defined by Dill [24] which originated with Rem, Snepscheut, and Udding [59]. We provide structural constructions and syntactic shorthands for labeled safe Petri nets that correspond to the behavioral semantics operations. Burch [14] extended trace theory semantics to timed circuits; we extend this work with an operational formalism that allows timing in the specification, and thus hierarchical timed verification.

Dill's trace theory is based on sequences of actions, but our nets allow transitions to be labeled with sets of actions. A trace theory based on sequences of sets of actions yields a conformance relation that distinguishes, for instance, interleaved and concurrent actions. In addition, composing a net that interleaves a pair of actions with another net that has those same actions labeling one transition may lead to an unintended deadlock. We do not attempt to resolve the complexities that arise in use of such a trace theory. Instead, we define conservative structural conditions on the use of labels consisting of sets of actions that allow us to use Dill's trace theory. For instance, we cannot perform verification using traditional trace theory on the instantaneous AND function block shown in Figure 64(b). However, when we compose that model with the simple buffer given in Figure 64(c) and hide the internal wire, the resulting net contains at most a single action for each transition and traditional trace theory can be applied.

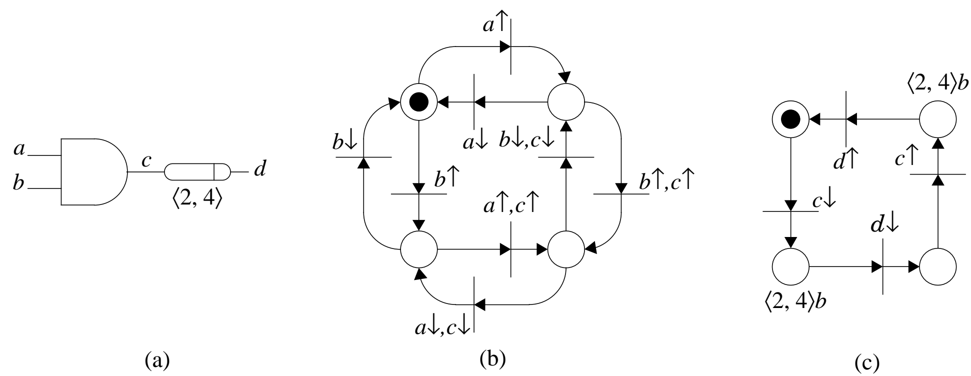


Figure 64: (a) AND gate with inputs a and b, and output d; (b) orbital net for functional behavior; (c) delay buffer with input c, output d, and delay of $\langle 2, 4\rangle$.

With these semantics, untimed constructions for receptiveness and synchronization apply unchanged to the timed case. Thus, implementing verification of trace structure conformance is straightforward. Determining whether an implementation conforms to a specification is reduced to determining if any of a specific set of failure transitions can be enabled. In addition, the trace theory operation of mirroring is also preserved, allowing hierarchical verification.

7.2 Generating the Orbital Net Representations

To verify that our synthesized timed circuits implement their timed specifications, our verification procedure begins with the timed HSE specification and the implementation given as a netlist of basic gates. To translate the specification to an orbital net representation, the same procedure described earlier is used except that the timing requirement for each behavior place in the preset of an output transition is changed to a constraint place with timing requirement $\langle 0, \infty \rangle c$. These constraints must be satisfied by the timed circuit implementation. Part of the specification orbital net for the SEL is shown in Figure 65.

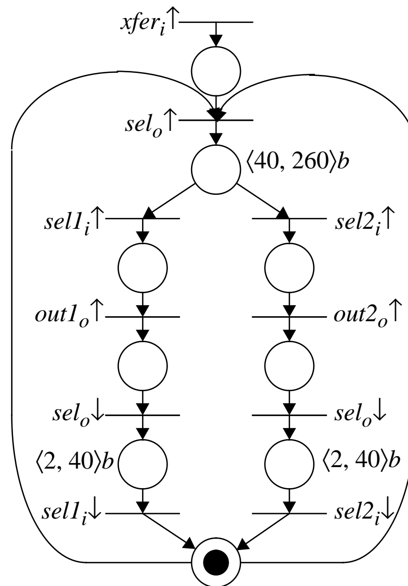


Figure 65: Part of the specification orbital net for the SEL.

For each gate in the implementation, an orbital net is constructed corresponding to an instantaneous function block such as the one given for the AND gate in Figure 64(b). This

net is composed with a delay element such as the one in Figure 64(c) with the behavioral timing requirement set by the delay given in the gate library. Each orbital net in the implementation is composed with the other orbital nets as dictated by the connections in the netlist.

7.3 Reporting Failures

To determine if a timed circuit implements its timed specification, the reachable state space is found using the partial order timing algorithm for the orbital net obtained by composing the implementation with its mirrored specification. If while exploring the state space a failure is detected, a sequences of transitions found using a depth-first search is reported that demonstrates the failure. This sequence, however, may be quite long, so after reporting the failure the procedure attempts to find a shorter sequence using a breadth-first search. Returning to the SEL, if we replace the standard C-implementation of $out2_o$ with the sum-of-products implementation as shown in Figure 66, it fails verification, and the following failure trace is reported:

$$xfer_i \uparrow, u_{26} \uparrow, sel_o \uparrow, data_o \uparrow, sel2_i \uparrow, data_i \uparrow, u_{26} \downarrow,$$
$$u_{32} \uparrow, out2_o \uparrow, u_{29} \uparrow, sel_o \downarrow, sel2_i \downarrow, u_{32} \downarrow, out2_o \downarrow$$

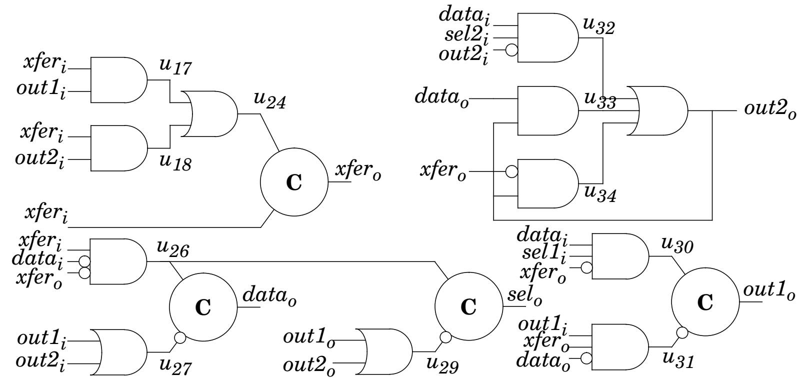


Figure 66: Implementation of the SEL which fails verification.

7.4 Verification Results

The verification procedure described in the previous section has been automated in the tool `Orbits` written by Tom Rokicki. This tool has been incorporated into the design system for timed circuits `ATACS`. Experimental results are given in Table 7 which were run on an HP9000/735 with 144 megabytes of memory using CScheme 7.3. The left four columns indicate values that are the same for geometric and partial order timing. The startup time is the time required to parse the input and construct the appropriate orbital net. The number of net nodes is the sum of the places and transitions in the resulting orbital net. The third column gives the number of untimed states. The fourth column gives the number of discrete states, after all timing parameters are divided by their greatest common divisor. The next four columns give the number of geometric regions and the run time in seconds for verification using standard geometric timing and partial order timing, respectively.

The first half of Table 7 consists of the automatically synthesized gate-level timed circuits described above. First, we find that the number of discrete states can be quite large making discrete-time verification difficult, if not impossible. Verification of these examples using partial order timing is also more efficient than the geometric timing approach. Especially in the case of the DRAM controller where the verification time is improved by over an order of magnitude.

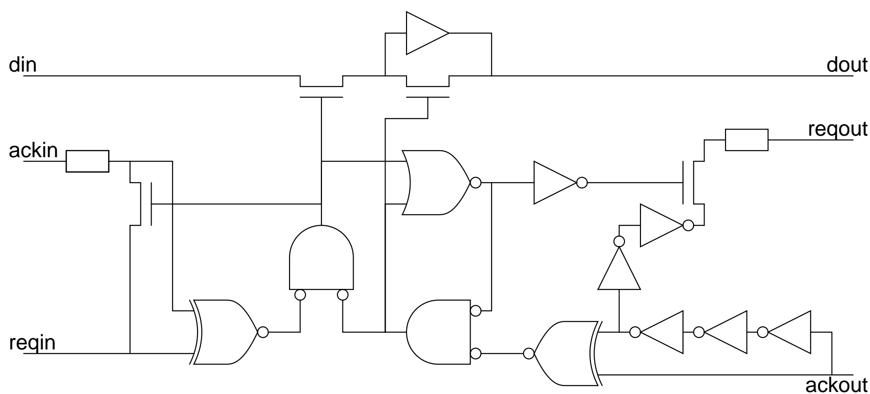


Figure 67: Seitz queue element.

The second half of the table consists of other timed circuits and systems that exhibit a high degree of concurrency. For example, the `seitz` queue element is pictured in Figure 67; `seitz2` is two connected copies of this circuit. The `kyy` examples [80] have thirty-seven gates and timing parameters given to three significant digits. Where the examples ran

out of time or space using the geometric method, often the verification was far from done. For the `seitz2` example, after one hour of CPU time, only 1,404 of the 4,572 untimed states have been seen, yet 473,202 distinct geometric regions have been encountered. One particular untimed state has 13,275 distinct geometric regions at this point. Partial order timing for this example finds the entire state space as 5,820 geometric regions in one half minute of CPU time.

Table 7: Verification results.

Examples	Startup time	Net nodes	Untimed states	Discrete states	Geometric regions	Geometric time	Partial order regions	Partial order time
SEL	2.59	770	271	6.16e5	582	1.91	358	1.76
SEL2	2.26	616	96	2033	130	0.33	102	0.29
MMU	5.94	2248	547	2.21e7	1163	5.22	583	2.03
DRAM	3.83	1326	8093	1.17e6	70611	1492.97	8899	98.13
TSBM	3.57	1464	305	49936	510	3.36	305	2.07
adv3x40	0.05	6	1	68921	1.52e5	164.99	1	0.01
adv4x40	0.03	8	1	2.83e6	out of memory		1	0.01
adv50x40	0.27	100	1	4.36e80	out of memory		1	60.21
phil3	0.19	149	144	27806	758	0.77	188	0.36
phil4	0.22	197	1152	9.82e5	out of time		1541	6.98
phil5	0.25	245	9840	3.47e7	out of time		14039	159.40
seitz	0.41	355	344	2.92e13	3234	5.48	416	1.22
seitz2	0.55	624	4572	5.48e19	out of memory		5820	29.79
kyy5	2.46	1484	5266	>1e20	out of memory		6083	56.74
kyy15	1.97	1484	18357	>1e20	out of memory		20250	321.47

Time values are given in seconds. An entry of *out of time* indicates that the verification did not complete within two hours, and an entry of *out of memory* indicates that the verification ran out of memory before completing.

One more thing to consider from Table 7 is the ratio of the number of regions found using partial order timing to the number of untimed states. We find that partial order timing often finds on average very close to one, and in all of our examples, no more than two geometric regions for every untimed state. This means that the partial order timing approach is achieving a near optimal representation of the timed state space.

Appendix

The verification procedure is implemented within the tool `Orbits`. After `ATACS` has synthesized a timed circuit implementation, it can be verified by `Orbits` using the command *verify* within `ATACS`. This command produces a file which includes both an orbital net specification and a description of the timed circuit implementation. If the timed circuit is gate-level, the delay information for the library of basic gates is read in from the file *library.ver*. After `Orbits` has completed the verification, it returns to `ATACS` and reports that either that it completed successfully or that the verification failed. If it fails, a sequence of signal transitions which exhibit the error are reported to the user.

Chapter 8

Conclusions

The purpose of a fish trap is to catch fish,
and when the fish are caught, the trap is forgotten;
the purpose of a rabbit snare is to catch rabbits,
and when the rabbits are caught the snare is forgotten;
the purpose of words is to convey ideas,
and when the ideas are grasped, the words are forgotten.
—Zhuang Zi

荃者所在魚
得魚而忘荃
蹄者所在荃
得兔而忘蹄
言者所在意
得意而忘言
一莊子

8.1 Summary

This thesis describes a methodology for the automatic synthesis and verification of gate-level timed circuits. To specify timed circuits, we created the timed HSE language which includes constructs for specifying sequencing, concurrency, and choice. The semantics of this specification language are defined using a new formal model, timed ER structures. We developed two timing analysis algorithms which can be used to obtain the reachable state space for the specification of the circuit being designed. The first is a heuristic algorithm for deterministic specifications. The second begins with a more general orbital net representation that is automatically obtained from the timed HSE specification, and it uses geometric regions and partial orders to efficiently represent and explore the timed state space. We also presented efficient algorithms for the synthesis of timed circuits which obtain a hazard-free timed circuit implementation using only basic gates, facilitating the use of semi-custom components. After obtaining an initial basic gate implementation, we map it to the given gate library using a technology mapping procedure based on resynthesis and

an iterative search guided by timing information. We demonstrated the effectiveness of the timed circuit design procedure on several practical examples, and our results indicate that our timed circuit implementations are significantly smaller and faster than those produced by other asynchronous and synchronous design methodologies. Finally, we verified all our timed circuits, and we showed that partial order timing verification can handle much larger, more concurrent examples than the standard discrete or geometric methods. By applying systematic methods that incorporate timing into asynchronous circuit design, our procedure produces both efficient and reliable implementations opening the door to the use of asynchronous circuits in domains previously dominated by synchronous circuits.

8.2 Future Work

While we believe that the results in this thesis show that timed circuits are a very promising alternative design style, there is much work that needs to be done in order to make it a truly viable alternative to existing synchronous design methods. This section briefly describes the areas that we believe to be the most important research problems which must be addressed.

8.2.1 Specification

Almost all commercial design tools for the simulation and synthesis of synchronous digital systems employ standard hardware description languages (either Verilog or VHDL). Current asynchronous specification methodologies (including ours described in Chapter 2) use non-standard languages such as CSP [44], OCCAM [13], or Tangram [69]. In order to take advantage of the excellent repertoire of existing tools and to make the transition to asynchronous design easier for designers, we believe that it is necessary in the future to develop methods and tools which use standard HDL's.

8.2.2 Compilation

After specifying a design at a *behavior-level* in a standard HDL, it must be compiled to a *register-transfer-level* (RTL) description, such as our timed HSE description. For every behavioral-level specification, however, there are a multitude of possible RTL descriptions, which make finding the optimal RTL description quite difficult. First, there are many ways to decompose a design into smaller, synthesizable blocks. Decomposition determines both the amount of global concurrency and the degree of pipelining, which are two major

factors in determining overall performance. Second, the asynchronous communications between blocks can be implemented in many ways including two-phase and four-phase handshaking. More aggressive communication methods can also be employed in a timed circuit that use only one wire for requests and infer the acknowledgment from circuit delays, analogous to synchronous communication. Finally, there are many possible alternatives to solving the state assignment problem. One method to do this is to automatically add state variables by extending a technique such as the one described in [78] to timed circuits. In addition, reshuffling of the placement of handshaking signals [44] or making tighter timing assumptions can also solve the state assignment problem by removing unnecessary concurrency.

8.2.3 Technology Mapping and Module Generation

As we saw in Chapter 5, significant improvements in area and delay can be achieved using generalized C-elements rather than standard-cells. We have found that many generalized C-element implementations can be mapped to precharged gates which are already found in existing cell libraries. Also, since generalized C-elements have a regular geometry, they can be automatically created using relatively straight-forward module generation techniques, as demonstrated by Alain Martin of Caltech. To get high-performance designs, we believe it is necessary to both explore mapping generalized C-element designs to cells found in existing libraries, as well as, developing techniques to automatically generate new cells.

8.2.4 Verification

In Chapter 7, we successfully verified all our timed circuit designs by using timing analysis techniques which efficiently abstract the timed state space, but state explosion of the untimed state space is still quite computationally challenging. Comparing the number of untimed states in verification with those found in synthesis, we see that the internal signals in the circuit implementations cause a significant increase in the state space size. To address this problem, we believe that the internal signal behavior can be abstracted by extending the cube approximation technique that is successfully used to reduce the complexity of verifying speed-independent circuits [4].

8.2.5 Asynchronous Datapaths

There are currently two major techniques for asynchronous datapath design, *bundled-data* and *dual-rail*. In the bundled-data approach a signal is transmitted with the data that indicates when the data is valid, much like the clock signal in a synchronous design. While this approach allows existing synchronous datapaths to be used, the need to delay the data valid for the worst-case delay precludes taking advantage of average-case, data-dependent delays. To achieve average-case performance, it is necessary to detect the completion of an operation. This is typically done by using dual-rail logic which uses two wires to encode both the positive and negative phase of the signal, as well as, when it is invalid. Fully dual-rail logic can have significant area and delay overhead which can often outweigh the advantage of average-case performance. For timed asynchronous datapath design, we believe an approach that combines the advantages of both bundled-data and dual-rail is necessary. One technique that we would like to explore is using *domino dual-rail logic* which uses skewed cones of domino logic stages in which common paths have fewer logic stages than less common paths. Dual-railed input signals ensure that the logic will be hazard-free, and domino logic stages facilitate a short reset time.

8.2.6 Interfacing with Synchronous Designs

Despite the advantages of asynchronous designs, they will not immediately replace all existing synchronous designs due to existing design expertise, and they may never replace synchronous designs in many domains. Systems in the future will have a mixture of synchronous and asynchronous modules which will need to communicate at very high rates. Therefore, it is necessary to develop methods to specify and design these *mixed-timed* modules. This thesis demonstrates that the timed circuit methodology can be used for fully asynchronous circuits, asynchronous circuits which interface with synchronous environments, and even synchronous circuits. While more research is necessary, we believe that the timed circuit design methodology will facilitate a smooth transition to the design of mixed-timed systems.

Bibliography

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

[2] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the Real-Time Systems Symposium*, pages 157–166. IEEE Computer Society Press, 1992.

[3] T. Amon, H. Hulgaard, G. Borriello, and S. Burns. Timing analysis of concurrent systems. Technical Report UW-CS-TR-92-11-01, University of Washington, 1992.

[4] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng. Efficient verification of determinate speed-independent circuits. In *Proceedings IEEE 1993 ICCAD* Digest of Technical Papers, pages 261–267, 1993.

[5] P. A. Beerel and T. H.-Y. Meng. Semi-modularity and testability of speed-independent circuits. *INTEGRATION, the VLSI journal*, 13(3):301–322, September 1992.

[6] P. A. Beerel and T. H.-Y. Meng. Logic transformations and observability don't cares in speed-independent circuits, 1993. In collection of papers of the *ACM International Workshop on Timing Issues in the Specification of and Synthesis of Digital Systems.*

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

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

[9] G. Borriello and R. H. Katz. Synthesis and optimization of interface transducer logic. In *Proceedings IEEE 1987 ICCAD* Digest of Papers, pages 274–277, 1987.

[10] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, 1984.

[11] R. K. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Proceedings IEEE 1989 ICCAD* Digest of Technical Papers, pages 316–19, 1989.

[12] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[13] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer-Aided Design, ICCAD-1989*. IEEE Computer Society Press, 1989.

[14] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon University, 1992.

[15] S. M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1987.

[16] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[17] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[18] T.-A. Chu. *Private Communication*, July 1991. Tam-Anh Chu is with Cirrus Logic.

[19] T.-A. Chu. Synthesis of hazard-free control circuits from asynchronous finite state machine specifications. *Journal of VLSI Signal Processing*, 7(1/2):61–84, February 1994.

[20] A. Davis, B. Coates, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, pages 409–418. IEEE Computer Science Press, 1993.

[21] M. E. Dean. STRiP: A self-timed RISC processor architecture. Technical report, Stanford University, 1992.

[22] M. E. Dean, D. L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (CSCD). *Journal of VLSI Signal Processing*, 7(1/2):7–16, February 1994.

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

[24] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.

[25] D. W. Dobberpuhl, R. T. Witek, R. Allmon, R. Anglin, R. Bertucci, S. Britton, L. Chao, R. A. Conrad, D. E. Dever, B. Gieseke, S. M. N. Hassoun, and G. Hoeppner. A 200 mhz 64 bit dual-issue cmos microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1155–1167, November 1992.

[26] J. C. Ebergen. *Translating Programs into Delay-Insensitve Circuits*. PhD thesis, Eindhoven University of Technology, 1987.

[27] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[28] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Transactions on Electronic Computers*, pages 350–359, June 1965.

[29] N. Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 333–346. Springer-Verlag, 1993.

[30] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[31] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *Proceedings of the 7th Symposium Logics in Computers Science*. IEEE Computer Society Press, 1992.

[32] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP 92: Automata, Languages, and Programming*, pages 545–547. Springer-Verlag, 1992.

[33] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, UK. LTD., Englewood Cliffs, New Jersey, 1985.

[34] D. A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Institute*, March, April 1954.

[35] H. Hulgaard and S.M. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.

[36] S. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *IEEE ICCAD Digest of Technical Papers*, pages 417–420, 1992.

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

[38] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *IEEE Transactions on Computer-Aided Design*, 14(1):61–86, January 1995.

[39] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[40] T. K. Lee. *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits.* PhD thesis, California Institute of Technology, 1995.

[41] H. R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical report, Harvard University, July 1989.

[42] K.-J. Lin, J.-W. Kuo, and C.-S. Lin. Direct synthesis of hazard-free asynchronous circuits from STGs based on lock relation and MG-decomposition approach. In *Proc. European Design and Test Conference (EDAC-ETC-EuroASIC)*, pages 178–183. IEEE Computer Society Press, 1994.

[43] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W. J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

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

[45] A. J. Martin, S. M. Burns, T. K. Lee, D. Borković, and P. J. Hazewindus. The design of an asynchronous microprocessor. In *Decennial Caltech Conference on VLSI*, pages 226–234, 1989.

[46] K. McMillan and D. L. Dill. Algorithms for interface timing verification. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

[47] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messershmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[48] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, Inc., 1985.

[49] C. J. Myers, P. A. Beerel, and T. H.-Y. Meng. Technology mapping of timed circuits. In *Asynchronous Design Methodologies*. IEEE Computer Society Press, May 1995.

[50] C. J. Myers and A. J. Martin. The design of an asynchronous memory management. Technical Report CS-TR-93-30, California Institute of Technology, 1993.

[51] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

[52] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[53] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis and verification of gate-level timed circuits. Technical Report CSL-TR-94-652, Stanford University, January 1995.

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

[55] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, 1994.

[56] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[57] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *International Conference on Computer Design, ICCD-1991*. IEEE Computer Society Press, 1991.

[58] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

[59] M. Rem, J. L. A. van de Snepscheut, and J. T. Udding. Trace theory and the definition of hierarchical components. In R. Bryant, editor, *Third Caltech Conference on VLSI*, pages 225–239. Computer Science Press, Inc., 1983.

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

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

[62] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[63] A. Shen, A. Ghosh, S. Devadas, and K. Keutzer. On average power dissipation and random pattern testability of CMOS combinational logic networks. In *Proceedings IEEE 1992 ICCAD* Digest of Papers, 1992.

[64] P. Siegel. *Automatic Technology Mapping for Asynchronous Designs.* PhD thesis, Stanford University, February 1995.

[65] P. Siegel, G. DeMicheli, and D. Dill. Automatic technology mapping for generalized fundamental-Mode asynchronous designs. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1993.

[66] P.A. Subrahmanyam. 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.* Springer-Verlag, 1990.

[67] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[68] S. H. Unger. *Asynchronous Sequential Switching Circuits.* New York: Wiley-Interscience, 1969.

[69] C.H. van Berkel and R. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *International Conference on Computer Design, ICCD-1988.* IEEE Computer Society Press, 1988.

[70] K. van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* PhD thesis, Eindhoven University of Technology, 1992.

[71] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[72] P. Vanbekbergen. *Synthesis of Asynchronous Controllers from Graph-Theoretic Specifications.* PhD thesis, Katholieke Unviversiteit Leuven, September 1993.

[73] P. Vanbekbergen, G. Goossens, and H. de Man. Specification and analysis of timing constraints in signal transition graphs. In *Proceedings of the European Design Automation Conference*, 1992.

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

[75] V. I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[76] T. E. Williams, M. Horowitz, R. L. Alverson, and T. S. Yang. A self-timed chip for division. In Paul Losleben, editor, *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 75–95. MIT Press, 1987.

[77] G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. Noordwijkerhout, Norway,* June 1988.

[78] C. Y.-C. and B. Lin. Optimised state assignment for asynchronous circuit synthesis. In *Asynchronous Design Methodologies.* IEEE Computer Society Press, May 1995.

[79] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarke. Efficient verification of parallel real-time systems. In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 321–332. Springer-Verlag, 1993.

[80] K. Y. Yun. Private communication, 1993.

[81] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems.* PhD thesis, Stanford University, 1994.

[82] K. Y. Yun and D. L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *Proceedings IEEE 1993 ICCAD* Digest of Papers. IEEE Computer Society Press, 1993.

[83] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *International Conference on Computer Design, ICCD-1992.* IEEE Computer Society Press, 1992.