CORRECTNESS AND REDUCTION IN TIMED CIRCUIT ANALYSIS

by

Eric G Mercer

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

Doctor of Philosophy

in

Electrical Engineering

Electrical and Computer Engineering

The University of Utah

December 2002

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Eric G Mercer

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

	Chair: Chris J. Myers
	Peter A. Beerel
	Ganesh Gopalakrishnan
	Reid R. Harrison
	Priyank Kalla

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

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

Date	Chris J. Myers Chair, Supervisory Committee

Approved for the Major Department

V. John Mathews
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

To increase performance, circuit designers are experimenting with *timed circuits*—a class of circuits that rely on a complex set of timing constraints for correct functionality. This is evidenced in published experimental designs from industry. Timing constraints are key to the success of these designs, and algorithms to verify timing constraints are required to make them practical in commercial applications. Due to the complexity of the constraints, however, traditional static timing analysis is not adequate. Timed state space analysis is required; thus, improved timed state space analysis is paramount to producing efficient timed circuits.

This dissertation discusses two facets of work in timed state space analysis: *correctness* and *reduction*. For correctness, this dissertation presents the *level-ruled Petri net* as a model for timed circuits. This model is based on the Petri net language. It includes, however, timing information and level expressions that are key to the specification and verification of timed circuits. This dissertation formalizes the intent of correctness in the verification of a timed circuit by defining a set of failure conditions that can be analyzed in the circuit's respective model. The circuit is said to be correct if its model is failure free. For reduction, this dissertation presents a timed state space analysis algorithm that verifies correctness in the timed circuit model. The algorithm, when compared to existing algorithms, reduces on average the running time and memory footprint of analysis. A partial order reduction is implemented for the algorithm to further reduce its resource usage. This reduction is not supported by the existing algorithms; thus, the new analysis algorithm can be applied to systems that are beyond their capacity. This is demonstrated in verifying industrial designs from IBM and Sun Microsystems.

To the joy of life and learning

CONTENTS

ABSTRACT . . . . **iv**

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

ACKNOWLEDGMENTS . . . . **xiii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Contributions 3
1.2 Overview 5

2. A TIMED CIRCUIT MODEL **6**

2.1 The Petri Net 8
2.1.1 Structure 8
2.1.2 Semantics 10
2.2 The Level-ruled Petri Net 12
2.2.1 Structure 14
2.2.2 Semantics 21
2.3 Examples 30
2.3.1 Elastic Synchronous Pipelines 30
2.3.2 Delayed-reset Domino Gates 32
2.3.3 STARI FIFO 35
2.4 Related Work 39
2.5 Summary 44

3. CORRECTNESS **46**

3.1 The Constraint Rule 47
3.1.1 Structure 51
3.1.2 Semantics 52
3.2 Correctness Properties 54
3.2.1 Safety 55
3.2.2 Consistent State Assignment 58
3.2.3 Output Semimodular 60
3.2.4 Constraint Satisfied 65
3.3 Correctness Statement 70
3.4 Related Work 72

3.5 Summary 73

4. TIMING ANALYSIS 75

4.1 Equivalence Classes 77
4.1.1 Zones 78
4.1.2 Timed State Class 86
4.2 Causal Group Set 89
4.3 Fireable Transitions 93
4.3.1 Causal Assigned Zone 96
4.3.2 Algorithm 103
4.4 Successor Transitions 108
4.5 Finite Representation 113
4.5.1 Timed State Class Graph 114
4.5.2 Algorithm 115
4.6 Pruning 118
4.6.1 Type I Redundant Transitions 119
4.6.2 Necessary Transitions 123
4.6.3 Type II Redundant Transitions 128
4.6.4 Algorithm 130
4.7 Timing Failures 132
4.8 Constraint Redundant 136
4.9 Exactness 137
4.10 Related Work 139
4.11 Summary 141

5. REDUCTION 143

5.1 Partial Order Reduction 144
5.1.1 Basic Notions 145
5.1.2 Basic Principles 146
5.1.3 Conditions 148
5.1.4 Ample Set Construction 150
5.2 Modular Synthesis 159
5.3 Related Work 163
5.4 Summary 165

6. RESULTS AND ANALYSIS 166

6.1 Timing Analysis 167
6.2 Reduction 175
6.3 Synchronous Interlocked Pipelines 179
6.3.1 Elastic Synchronous Pipelines 181
6.3.2 Two-phase Interlocked Pipelines 183
6.3.3 Master-slave Interlocked Pipelines 184
6.3.4 Fork 186
6.3.5 Branch 186
6.3.6 Join 187
6.3.7 Priority Select 189

6.4 Summary 189

7. CONCLUSIONS AND FUTURE WORK 191

7.1 Specification 191
7.2 Correctness 192
7.3 Timing Analysis 193
7.4 Reduction 196

APPENDICES

A. OUTPUT SEMIMODULAR TRANSITIONS 197

B. BOOLEAN FUNCTION MANIPULATION 202

REFERENCES 208

LIST OF TABLES

6.1 Comparison with POSET Timing on Regression Suite I 169
6.2 Comparison with POSET Timing on Regression Suite II 170
6.3 Comparison with POSET Timing on Dissertation Examples 171
6.4 Comparison of Algorithms at Limits 175
6.5 Cost of Modular Analysis 176
6.6 Limits for Modular Analysis 177
6.7 Comparison to Abstraction and VINAS-P 178

LIST OF FIGURES

2.1 Two contrasting representations of the function $c = a \wedge b$. 7

2.2 A graphical representation of a simple Petri net. 9

2.3 The Muller C-element and its defined behavior. 17

2.4 A Muller C-element level-ruled Petri net model. 18

2.5 A net fragment showing a transition with two rules. 22

2.6 Net fragments illustrating the maximum delay definition. 26

2.7 A gate schematic of the IBM elastic synchronous pipeline. 31

2.8 The latch and environment models for the elastic pipeline. 31

2.9 The replication and mapping for the specification of the second stage. 33

2.10 A two-stage delayed-reset domino gate to compute $(a \vee b) \wedge c$. 33

2.11 The net models for the delayed-reset domino gate. 34

2.12 Three environment models for the delayed-reset domino gate. 35

2.13 The block diagram of a dual-rail STARI FIFO with two stages. 35

2.14 The environment specification for the STARI FIFO. 37

2.15 The empty and full stage model for a STARI FIFO. 38

2.16 The level-ruled Petri net for the composed STARI FIFO. 39

2.17 Two representations of the function $c = a \wedge b$. 42

2.18 Conflict structures in the Petri net and timed event/level models. . . . 43

3.1 The block diagram of a dual-rail STARI FIFO with two stages. 48

3.2 The empty and full stage model for a STARI FIFO with constraints. . 49

3.3 A two-stage delayed-reset domino gate and its *clk2* model. 50

3.4 The level-ruled Petri net for the composed STARI FIFO. 56

3.5 The second stage level-ruled Petri net model to compute f_2. 61

3.6 A net fragment of a composition with a member component interface. 61

3.7 A net fragment with its full and reduced state space. 64

3.8 A net fragment that is not constraint satisfied. 67

3.9 The level-ruled Petri net for the STARI FIFO with constraints. 68

4.1 A zone, its canonical form, and a superset relation. 83
4.2 A zone showing two concurrent transitions from a set of transitions. . . 86
4.3 A fragment of a closed system with a portion of its initial zone. 88
4.4 A fragment with a merge place and a rule with a Boolean function. . . 89
4.5 A level-ruled Petri net fragment with four potential fireable transitions with its zone. 94
4.6 A causal assigned zone and its canonical form. 96
4.7 A level-ruled Petri net fragment with three potential fireable transitions with its zone. 100
4.8 An algorithm to create causal assigned zones. 102
4.9 A series of three zones that show creation of a causal assigned zone. . . 103
4.10 The two causal zones for the t_7 causal assignment. 104
4.11 An algorithm that computes fireable transitions. 104
4.12 The evolution of a zone in the fireable algorithm as transitions are added. 107
4.13 A causal assigned zone and its canonical form. 108
4.14 An algorithm that computes successor timed state classes. 109
4.15 A causal assigned zone for transition t_6 on t_2 and its canonical form. . 111
4.16 A causal assigned zone for transition t_6 on t_1 and its canonical form. . 112
4.17 A causal assigned zone that is not valid. 112
4.18 An algorithm to find a timed state class graph. 117
4.19 A level-ruled Petri net, zone, and graph to illustrate pruning. 119
4.20 Two causal assigned and a reduced zone for Fig. 4.19. 121
4.21 A series of zones to demonstrate pruning on marking and level satisfied transitions. 123
4.22 Two nets with transitions that are not enabled. 124
4.23 Two level-ruled Petri nets that can generate multimember necessary sets. 125
4.24 An algorithm to compute a set of necessary transitions. 126
4.25 A level-ruled Petri net, zone, and graph to illustrate pruning requirements. 128
4.26 An algorithm to prune transitions from the timed state class. 130
4.27 A level-ruled Petri net, graph, and zone to illustrate a timing failure. . 133
4.28 An algorithm to check timing failures on constraint rules. 136

4.29 Two level-ruled Petri nets with a zone that create conservative zones in the reachable state space. 138
4.30 Two causal assigned zones in their canonical form showing fireable transitions. 139
5.1 Two level-ruled Petri nets that contain transitions that can be disabled. 152
5.2 Level-ruled Petri nets that show order dependent failures. 154
5.3 Two level-ruled Petri nets with conflicting transitions that are not fireable together. 157
5.4 A simple timed circuit with its level-ruled Petri net model. 161
5.5 The graph of the edge set and its reduced form. 161
6.1 The *beta* example with two signals a and b. 172
6.2 Elastic synchronous pipeline. 182
6.3 Two-phase interlocked pipeline. 183
6.4 Hazard free circuit for stall signal. 184
6.5 Master-slave interlocked pipeline. 185
6.6 A one to two fork implementation. 187
6.7 A one to one-of-two branch implementation. 188
6.8 A two to one join implementation. 188
6.9 A one-of-two to one priority select implementation. 189
A.1 A net fragment with its full and reduced state space. 200
B.1 A fragment with a merge place and a rule with a Boolean function. . . 203

ACKNOWLEDGMENTS

No work is the result of a single person. We are all supported and influenced by those around us. This is the case with the research in this dissertation. It is the culmination of the research efforts of several people. There are, however, key individuals that I would like to acknowledge. Their support has made a direct and significant impact on this work. Each is a cornerstone in the foundation of my academic career.

My family is the key cornerstone. They have toiled by my side these many long years of education. My wife Shennon never waivers in her support, and her patience is proven to be near limitless. Our children are a continual well of joy that helps push through the hard times. My extended family is an important factor too. They do not hesitate to give their time and means in helping us find our way to the end of this journey. I humbly thank them for all they do, and I am grateful that they are part of my life.

Chris J. Myers is the next cornerstone in my foundation. Although Chris is my advisor, he is also my friend. His insight is invaluable, and his efforts to further this work are continual. Chris started me on this path in 1996 when he brought me into his lab. I would not be writing this without Chris' financial, emotional, and intellectual support. He is an example to me both as a good person and successful academic. I am grateful that Chris is part of my life.

My colleagues are the third cornerstone in my foundation. I worked directly and indirectly with many gifted people in doing this research. I want to first thank Tomohiro Yoneda. Not only did he introduce us to the idea of pruning in zones, but he has been open with his work in helping us understand his algorithms. I want to next thank Eric Peskin. Support for arbitrary Boolean functions in the algorithm derived from one of his many ideas, and he built and supports all of our LaTeX infrastructure. I want to thank Wendy Belluomini who brought me into IBM for a brief period. She broadened my vision and helped formulate many ideas. I want to thank the folks in our research group. I have benefited from my association with them. I want to finally thank other researchers in this area. Reading their work provided many pieces of this research. The key piece is produced by Marius Minea. I thank him for publishing his idea for an improved POSET algorithm in his Ph.D. dissertation. It is the starting point for this dissertation.

I want to thank each member of my supervisory committee. Peter Beerel has made significant contributions to this work both in its content and presentation. He is an invaluable resource. Ganesh Gopalakrishnan has provided a wealth of related work and readings. He is exceptionally well read and has guided me to key papers related to this area. I thank Reid Harrison for asking questions related to the practical use of this work, as well as helping me focus my future work in this area. Finally, I want to thank Priyank Kalla. He started his new position at the University of Utah, moved his family, and served on my committee all at the same time. He opened new directions for this work, and provided invaluable feedback on the manuscript. I am grateful for the combined efforts of the supervisory committee in bringing this work to fruition.

The fourth and final cornerstone is the joy of learning. Learning pervades all facets of life and is the component that makes life interesting. All things that people do, in a basic sense, are learning experiences. I thank the University of Utah for creating a safe environment where learning can take place. I also thank the staff and faculty in the Department of Electrical and Computer Engineering. I have a never ending appetite for learning. This research is an expression of that appetite made by possible by this department and institution; as well as the many people who make it a reality.

CHAPTER 1

INTRODUCTION

The tools and methodologies used in current design flows have evolved over many years. They have proven themselves in bringing to market many successful products. Although these tools often undergo incremental changes and modifications, for the most part, they remain intact from one development cycle to another. The important issue, however, is that the tools support a single type of design methodology. If the methodology changes, then the tool support is severed. This can be the first step to a failed product cycle for the engineering team. The biggest challenge in any design cycle is managing the complexity of the design. This task cannot be managed by hand, nor can it be understood by any one person.

The reality of current process technology is driving designers to experiment with alternative circuit architectures. The primary impetus for this change of style is power and noise. The current of change, however, is met with fierce opposition. The opposition is rooted in money. There is a large amount of money involved in any design project. The risk involved in bringing a commercial product to market is not taken lightly. For this reason, many product managers are reluctant to try new technology despite growing issues in current design practices; thus, change is only found in a development cycle when forced by the needs of the application.

If profit drives the design cycle, then verification is a real threat to profit. It is not uncommon to expend fifty percent of the development resources on verification alone. The cost of a product reaching the market with a critical defect is staggering. Intel demonstrated this reality with their floating point debacle. Not only was the Intel name marred, but the company stock had to weather a rather unpleasant downturn. This type of cost is staggering; thus, it is imperative that designs are thoroughly and completely verified before they reach the market. They must be free of critical design defects—defects that can be observed by the end consumer.

Functional validation is not sufficient to verify a design. It is not possible to run all possible test vectors through a design. This is not the only issue. It is not possible to check a circuit in all possible timing configurations either. The new circuits being forced into designs are often highly timed. Timing tools are common in any design methodology. These tools check that there are no errors in the fast and slow paths of the circuits. The new circuits, however, may have errors that result from the interaction of the fast and slow paths. This is not checked by common timing tools. These circuits do not lend themselves to traditional validation approaches.

Timed circuits are a class of circuits that rely on a complex set of timing constraints for correct functionality. A timed circuit can effectively address power and noise issues in modern design. This is evidenced in published industrial scale experimental designs [1, 2, 3, 4]. These designs, however, lack formal design methodology and tool support because they are experimental. Satisfying timing constraints is key to the success of these designs; thus, algorithms to check timing constraints are required to make them practical in commercial applications. Due to the complexity of the timing constraints, however, traditional static timing analysis is not adequate. Timing failures are a dynamical property of timed circuits. Timed state space exploration is required; thus, improved timed state space exploration is paramount to bringing this technology into mainstream design.

Academia has developed several solutions to the verification challenge. These solutions, however, have proven impractical in real design. The goal of this dissertation is to apply correctness and reduction to the analysis of industrial scale circuits in an effort to show that timed circuit analysis can be pragmatic. The presented circuits have appeared in various publications and evidence the forced hand of industry to address new process issues. Although industry has, for the most part, put in place tool support for these design styles, the tool path can be improved. The approach presented in this work is not a solution to the verification and synthesis problem in new architectures—there is much left to be done—but it is a step forward from previous technology, and it demonstrates the potential of timed circuit analysis.

This dissertation presents a timed state space exploration algorithm for timed circuit analysis. The goal is to develop the necessary algorithms to support new timed circuit styles that address growing process needs. It first presents the level-ruled Petri net as a timed circuit model. The level-ruled Petri net combines the event structure of the Petri net with the logic structure of a state machine. The end result is a model the lends itself to timed circuit analysis. This dissertation then discusses correctness in the model. It formally defines failure conditions, and it defines the meaning of correctness in a model of a circuit to help designers better understand the results of analysis. The dissertation next presents a new timed state space exploration algorithm to validate correctness in a level-ruled Petri net. The algorithm supports arbitrary expressions in the model, and reduces running time and the representation size of the timed state space on average.

The timed state space is too large in real world designs. Although the new algorithm improves the situation, it does not solve the issue. Timed circuit technology cannot be brought into the mainstream until it can address larger designs. This dissertation presents a reduction method for the timed state space exploration algorithm based on partial order reduction. The reduction enables modular analysis of large systems. The result is exact modular synthesis using partial order reduction. The effectiveness of the reduction is demonstrated in the analysis, synthesis, and verification of several industrial scale designs. Although the explosion of the timed state space is not contained by this work, it is better managed; thus, it takes timed circuits one step closer to mainstream design.

1.1 Contributions

This dissertation makes four specific contributions to the analysis of timed circuits: a new timed circuit model, a formal definition of correctness in the model, a timed state space exploration algorithm that supports arbitrary Boolean functions, and a partial order reduction on this model for verification and exact modular synthesis.

The first contribution is the level-ruled Petri net. It combines the event structure of the Petri net with the logic structure of a state machine. A transition is now governed by the marking, the Boolean state, and time. This is suited to timed circuit specification because the Boolean functions in the model create a form of syntactic abstraction that simplifies the drawn model structure. Standard and nonstandard gates can be compactly modeled by the level-ruled Petri net using their logic structure. The model, however, preserves the event behavior of a Petri net; thus, distinct signal transitions can be validated as the model moves through states. Circuit level effects can be adequately modeled too. The level-ruled Petri net is a modular language. Timed circuits can be specified modularly in higher level languages and then modeled at the low level by the level-ruled Petri nets with well-defined interfaces. The compact specification facilitates efficient analysis.

The second contribution is a formal definition of correctness in a level-ruled Petri net model of a timed circuit. Timing analysis has no meaning without an understanding of correctness. The designer must know exactly the behaviors being considered in the analysis. The designer must also know what a correct result means. The model of a timed circuit is correct if it is safe, consistent state assigned, output semimodular, and constraint satisfied in this work. These properties are defined in the level-ruled Petri net semantics.

The third contribution is a new timing analysis algorithm to validate correctness in the level-ruled Petri net model of a timed circuit. The algorithm supports arbitrary Boolean functions in the model and implements a partial order reduction on the timing information. The algorithm has better running time and builds a smaller representation of the timed state space on average when compared to prior work. It is the first published timing analysis algorithm that supports arbitrary Boolean functions; thus, simplifying model specification.

The fourth, and final, contribution is a partial order reduction for the level-ruled Petri net that supports syntactic abstraction. The reduction can also preserve the exact state space of a module in a larger system. This facilitates exact modular synthesis of a component in a larger system. The reduction does not require the designer to modify the system model, but uses a partial order reduction to avoid exploring the firing orders of independent signals. This significantly improves the running time of the timing analysis. It extends timing analysis and state based synthesis methods to systems that are larger than previously possible. This is demonstrated in the analysis, verification, and synthesis of several industrial designs.

1.2 Overview

The dissertation is organized to follow its contribution list. Each chapter addresses a contribution. Detailed related work specific to each contribution is given in its corresponding chapter for a clear perspective of this work. As such, Chapter 2 is devoted to the level-ruled Petri net as a timed circuit model with its structure and semantics. Chapter 3 is correctness. Each correctness property is defined and given context to a circuit level perspective. Chapter 4 is the new timing analysis algorithm for the level-ruled Petri net. Chapter 5 is the partial order reduction in the level-ruled Petri net. The reduction preserves the state space of the component module; thus, state based synthesis techniques can be used to modularly synthesize complete systems. Chapter 6 is a set of results and case studies. It applies the new modular analysis to several academic and industrial designs. The goal of the chapter is to expose both the strengths and weaknesses of the modular analysis. Chapter 7, finally, concludes the dissertation by summarizing the impact of the contributions and presenting future work in this area.

CHAPTER 2

A TIMED CIRCUIT MODEL

The timed circuit model defines the capabilities of a CAD application. This follows from the realization that an algorithm can only operate on behaviors represented in the model. If a model does not capture a certain behavior of the system, then that behavior cannot be analyzed or discovered. If the model represents too many behaviors, however, then the cost of analysis becomes prohibitive; thus, a model must be carefully designed to meet the needs of the particular application.

A timed circuit model must represent all possible transition times of signals in the system, and it must be approachable by a designer. The first requirement stems from the goal of timed circuit analysis to establish correctness in the circuit. To this end, it is necessary to know when transitions can happen in the system, and an analysis algorithm verifies that transitions not only occur, but that they do not occur at bad times. The second requirement is more pragmatic. If the model for a circuit cannot be intuitively composed directly, or derived from, a higher level language, then it becomes too difficult to specify complex systems. A timed circuit model must therefore capture all transition times of important signals in the system using a simple intuitive structure.

The level-ruled Petri net addresses the needs of a timed circuit model. It is based on the widely accepted Petri net, but it augments the Petri net with a notion of time and logic. The need for time in the Petri net model is self-evident in a timed circuit application. The need for logic, however, is best understood through illustration. Fig. 2.1(a) is a Petri net model of the function $c = a \wedge b$. Although the semantics of the Petri net model have yet to be presented, the complexity of Fig. 2.1(a) evidences that Petri nets do not readily lend themselves to simple logic functions. The level-ruled Petri net model in Fig. 2.1(b) of the same function is a stark contrast to its Petri net compatriot. The ability to specify logic functions decreases the structural complexity of the net and makes the model more approachable by designers. Although this reduction increases the complexity of analysis, it is not enough to nullify the decrease in structural complexity.

The goal of this chapter is to present a comprehensive semantic model of the level-ruled Petri net and to give an intuitive appreciation for its expressive power, as well as its limits. To this end, Section 2.1 presents the Petri net as the theoretical basis for defining concurrent systems. The level-ruled Petri net augments the expressiveness of the Petri net, so it is useful to understand the Petri net semantics. Section 2.2 is the formal presentation of the level-ruled Petri net as a model for timed circuits with its underlying structure and semantics. An intuitive appreciation of this new model is gained in Section 2.3; it demonstrates the use of the level-ruled Petri net in the specification of several timed systems. Section 2.4 is a retrospective look at other methods of modeling circuits. It tries to bring perspective to the level-ruled Petri net with regard to other approaches. This chapter is concluded in Section 2.5 with a brief summary of the presented material as it relates to correctness and reduction in timed circuit analysis.

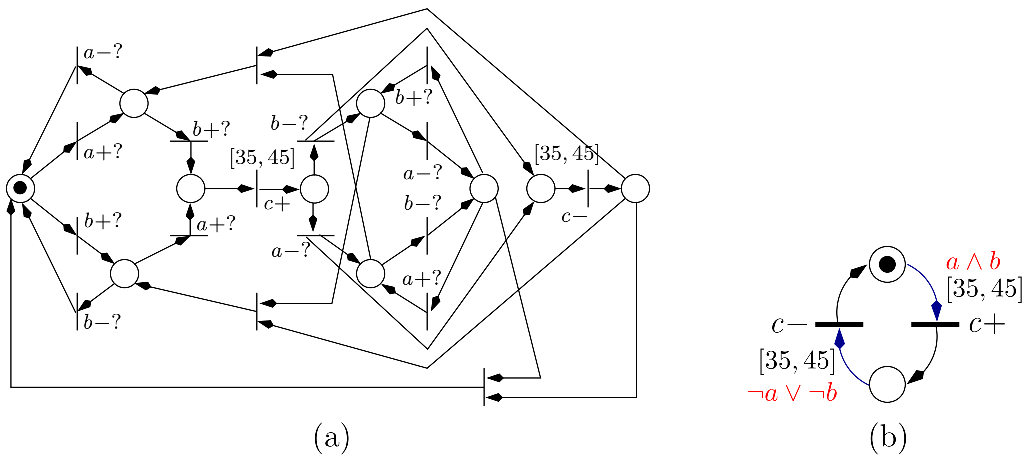


Fig. 2.1. Two contrasting representations of the function $c = a \wedge b$. (a) The Petri net representation. (b) The level-ruled Petri net representation.

2.1 The Petri Net

The Petri net is a studied, understood, and formalized language to model systems with concurrency [5]. It has been successfully employed to model and then analyze a myriad of systems. Although it does not include a notion of time in its ordinary form, it does provide infrastructure to model a timed circuit as a grouping of discrete parallel systems. In this sense, each component of the circuit is an independent agent that communicates with other agents in the circuit. The interface at each agent is precisely defined by a set of allowed traces. A *trace* is an ordered vector of transitions, and the Petri net is a graphical representation of a trace set; thus, the allowed trace set for each component can be defined by an appropriate Petri net. This section briefly describes the safe Petri net that is a restricted form of the ordinary Petri net. A more complete discussion of the Petri net is given in [6, 7, 8]. This section is divided into two parts: the first part deals with the structure and graphical representation of the safe Petri net; and the second part presents the formal semantics of the structure and illustrates how it defines the interface of a component. The employed notation is largely based on that presented in [8].

2.1.1 Structure

The *Petri net* is a directed bipartite digraph on transitions and places. Its mathematical model is the four-tuple presented in Definition 2.1.

Definition 2.1 (Petri net Structure). *A Petri net is represented by the four-tuple $N = (T, P, F, \mu_o)$ that defines its transitions, places, flow relation, and initial marking.*

The first node set of the digraph is T, the finite set of transitions. The second node set is P, the finite set of places. The connectivity between the transitions and places is defined by the flow relation, $F \subseteq (T \times P) \cup (P \times T)$. It is a subset of all possible connections between members of the transition and place sets. The bipartite property of the digraph forces connectivity to only exist between the two node sets; transitions are always separated by places, and places are always separated by transitions. Because the graph is directed, these connections are in one direction only. The final member of the Petri net four-tuple in Definition 2.1 is the initial marking μ_o. A *marking* is any subset of places, $\mu \subseteq P$, and it defines the complete state of the Petri net. The *initial marking* μ_o is the initial state of the Petri net.

A graphical illustration of a simple Petri net is shown in Fig. 2.2. The interface behavior defined by this net is presented in Section 2.1.2. The figure is presented here to elucidate the structure of the Petri net. The thick lines in the graph represent transitions from the transition set $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$. The open circles denote places from the place set $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$. The arcs that connect the places and transitions belong to the flow relation. The flow relation for this graph is

$$F = \left\{ \begin{array}{llll} (t_1, p_1), & (t_1, p_2), & (p_1, t_2), & (p_2, t_3), \\ (t_2, p_3), & (t_3, p_4), & (p_3, t_4), & (p_4, t_4), \\ (t_4, p_6), & (t_4, p_5), & (p_5, t_5), & (p_6, t_6), \\ (t_5, p_7), & (t_6, p_8), & (p_8, t_1), & (p_7, t_1) \end{array} \right\}, \tag{2.1}$$

where each member of the flow relation is an ordered place-transition or transition-place pair. The filled circles in the figure are tokens. They indicate members of the marking set $\mu_o = \{p_1, p_2\}$. This is the initial state of the net. From this state, all possible future states of the net can be computed. This is the topic of the next section.

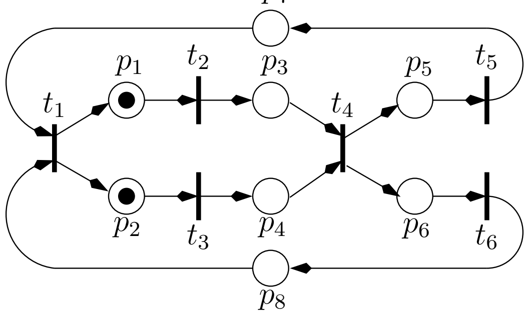


Fig. 2.2. A graphical representation of a simple Petri net.

2.1.2 Semantics

The state of the Petri net is defined by its marking. This state can change by firing a satisfied transition from the marking. The *preset* of a transition is the set of places that feed into it. For a transition $t \in T$, this is defined as $\bullet t = \{p \in P \mid (p,t) \in F\}$. Note that a similar set is defined for a place: $\bullet p = \{t \in T \mid (t,p) \in F\}$. A transition is *marking satisfied* if the members of its preset form a subset of the places in the marking. This is shown in Definition 2.2.

Definition 2.2 (Marking Satisfied Transition). *The transition t is marking satisfied by μ if $\bullet t \subseteq \mu$.*

For the Petri net in Fig. 2.2, the marking is $\mu = \{p_1, p_2\}$; the presets for transitions t_2 and t_3 are $\bullet t_2 = \{p_1\}$ and $\bullet t_3 = \{p_2\}$. The two transitions t_2 and t_3 are marking satisfied in μ because $\bullet t_1 \subseteq \mu$ and $\bullet t_2 \subseteq \mu$.

Firing a marking satisfied transition updates the marking to reflect the new state of the system. The *postset* of a transition is the set of places that the transition feeds into. For a transition $t \in T$, this is defined as $t\bullet = \{p \in P \mid (t,p) \in F\}$. Note that a similar set is defined for a place: $p\bullet = \{t \in T \mid (p,t) \in F\}$. Firing a transition in a marking removes from the marking any places in the preset of the transition, and it adds to the marking any places in the postset of the transition. This process is formalized in Definition 2.3.

Definition 2.3 (Marking Update). *The marking created from firing a transition t that is marking satisfied in μ is computed as $\mu' = \mu - \bullet t + t\bullet$.*

The notation $\mu\,[t\rangle\,\mu'$ indicates that the firing of t from μ leads to μ', where μ' is the marking given by Definition 2.3 as applied to μ and t. For the Petri net in Fig. 2.2, the transitions t_2 and t_3 are marking satisfied in $\mu = \{p_1, p_2\}$. If t_2 is fired from μ as indicated by $\mu\,[t_2\rangle\,\mu'$, then by Definition 2.3, μ' is given as

$$\begin{aligned} \mu' &= \mu - \bullet t_2 + t_2\bullet \\ &= \{p_1, p_2\} - \{p_1\} + \{p_3\} \\ &= \{p_2, p_3\}\,. \end{aligned}$$

If, however, the transition $\mu\,[t_3\rangle\,\mu'$ is taken instead, then Definition 2.3 computes μ' as $\{p_1, p_4\}$.

A *firing sequence* is a vector of transitions where each transition leads to a marking where the next one can fire. This is expressed in Definition 2.4.

Definition 2.4 (Firing Sequence). *The vector $(t_1, t_2, \ldots, t_{n-1}, t_n)$ is a firing sequence for a Petri net if there exists a vector of markings $(\mu_0, \mu_1, \mu_2, \ldots, \mu_{n-1}, \mu_n)$ such that the following holds: t_i is marking satisfied in μ_{i-1} and $\mu_{i-1}\,[t_i\rangle\,\mu_i$ for $1 \leq i \leq n$.*

The vector (t_2, t_3, t_4) is a firing sequence for the net in Fig. 2.2 because the vector of markings $(\{p_1, p_2\}, \{p_2, p_3\}, \{p_3, p_4\}, \{p_5, p_6\})$ satisfies Definition 2.4. Transition t_2 is marking satisfied in $\{p_1, p_2\}$ and firing it leads to $\{p_2, p_3\}$ where t_3 is marking satisfied. Firing t_3 leads to $\{p_3, p_4\}$ enabling t_4. Firing t_4 ends the firing sequence in $\{p_5, p_6\}$. The vector (t_2, t_4, t_3), however, is not a firing sequence because there does not exist an vector of markings that satisfies Definition 2.4.

One marking is *reachable* from another marking if there exists a firing sequence that takes the net from the one marking to the other. This is expressed in Definition 2.5.

Definition 2.5 (Reachable). *The marking μ' is reachable from μ if there exists a firing sequence that starts in μ and ends in μ'.*

Similar to the notation for Definition 2.3, $\mu_o\,[\mathbf{t}\rangle\,\mu_n$ indicates that μ_n is reachable from μ on the firing sequence $\mathbf{t}$, where $\mathbf{t}$ indicates a vector of transitions. The marking $\mu = \{p_5, p_6\}$ is reachable from the marking $\mu_o = \{p_1, p_2\}$ shown in Fig. 2.2 because of the firing sequence $\mathbf{t} = (t_2, t_3, t_4)$ from the previous example; thus, $\mu_o\,[\mathbf{t}\rangle\,\mu$. The marking $\mu = \{p_1, p_5\}$ is not reachable from μ_o because no firing sequence starting at μ_o exists to create μ.

The set of *reachable markings* defines the behavior of the Petri net. This set is denoted by $[\mu_o\rangle$ and is a subset of the power set of places P from the Petri net.

Definition 2.6 (Reachable Markings). *The reachable markings from μ is the set of markings $[\mu\rangle \subseteq 2^P$ where $\mu' \in [\mu\rangle$ if either $\mu' = \mu$ or there exists a firing sequence $\mathbf{t}$ such that $\mu\,[\mathbf{t}\rangle\,\mu'$.*

This set is finite as shown above because the marking is a subset of places in the net.

Definition 2.7 (Safe). *A Petri net is safe if for each $\mu, \mu' \in [\mu_o\rangle$ and for each transition t that is marking satisfied in each μ the following holds: $\mu\,[t\rangle\,\mu' \implies (\mu - \bullet t) \cap t\bullet = \emptyset$.*

A set of marking satisfied transitions can be created for a reachable marking of the net. In firing any of these transitions from the marking, the marking update in Definition 2.3 must never add a place to the marking that already exists in the marking. The net is safe if this property holds for every reachable marking. The Petri net in Fig. 2.2 is a safe Petri net. The set of reachable markings from the initial marking $\mu_o = \{p_1, p_2\}$ is

$$[\mu_o\rangle = \left\{ \begin{array}{cccc} \{p_1,p_2\}, & \{p_1,p_4\}, & \{p_2,p_3\}, & \{p_3,p_4\}, \\ \{p_5,p_6\}, & \{p_5,p_8\}, & \{p_6,p_7\}, & \{p_7,p_8\} \end{array} \right\}.$$

This net has eight reachable markings. The reachable marking set can be created from a depth-first search of all marking satisfied transitions starting from the initial marking. The depth-first search algorithm forms the basis of Petri net analysis.

2.2 The Level-ruled Petri Net

The level-ruled Petri net is an extension of the Petri net. The goal of the extension is three fold: first, to create a mapping between the Petri net and the physical circuit; second, to add a notion of time to the Petri net; and third, to simplify the specification process and resulting model structure through syntactic abstraction. The level-ruled Petri net is not meant to replace the Petri net, but to amend the Petri net to the analysis of timed circuits.

A function of the level-ruled extension is to map the Petri net to a physical model. Various members of the extension define the inputs and outputs to a circuit and provide the initial values for all output wires in the specification. The level-ruled Petri net is designed to allow the timed circuit to be divided into small manageable components. Large complex designs are composed of small easy to understand components. The collective behavior of the components is analyzed to see if errors exist in the system.

Time is added to the Petri net with the creation of the rule. A *rule* is a timing and level annotation on an edge between a place and a transition. The rule affects the firing semantics of the transition. A marking satisfied transition can no longer fire immediately from a marking. The transition must be marking satisfied and have its level information satisfied. Once these two conditions are met, it then waits for its timing information to be satisfied after which it can fire. A transition must fire before all of its timing information expires. The rule is a means to model an arbitrary delay on a transition. It is also a means to simplify the structure of the net.

The level information contained in the rules is a syntactic simplification of the Petri net. It simplifies the net's structure. Another benefit of the syntactic abstraction in levels is the ease of specification. The structure of the level-ruled Petri net using functions is approachable by hand—even for sizable specifications. More importantly, however, is that it can be compiled to from higher level languages such as timed handshaking expansions and VHDL [9, 10]. This can be a tremendous asset to ameliorating the design flow.

This section is organized like Section 2.1. The structure of the level-ruled Petri net is presented with a simple example. This is followed by the semantic definition. The presentation does not redefine any of the Petri net semantics from Section 2.1; rather, it augments the definitions from that section to shape the semantic behavior of the level-ruled Petri net. The goal of this new model is to build on the Petri net and its terminology.

The notion of a rule is first defined before the structure and semantic definition of the level-ruled Petri net is presented. A *rule* is a place-transition pair in the flow relation F for which timing bounds and Boolean functions can be defined; the set of all rules is given as $R = F \cap (P \times T)$. The rules essentially modify the semantic behavior of transitions in the Petri net from Definition 2.9. To simplify the presentation, a function to return the rules for a transition is defined.

Definition 2.8 (Rule Set). *For any transition $t \in T$, the rule set of t is given as $R(t) = \{(p, t) \in R \mid p \in \bullet t\}$.*

The rule set of transition t is now given by $R(t)$. It is the set of rules defined by the places in its preset.

2.2.1 Structure

A level-ruled Petri net is a Petri net coupled with a level-ruled extension as shown in Definition 2.9.

Definition 2.9 (Level-ruled Petri net). *A level-ruled Petri net is the pair $M = (N, E)$ consisting of a Petri net $N = (T, P, F, \mu_o)$ and its level-ruled extension E; the Petri net is such that for all pairs of transitions $(t, t') \in T$ where $t \neq t'$, $|t\bullet \cap \bullet t'| \leq 1$.*

The connectivity in the Petri net is restricted to not allow more than a single rule between two transitions. This restriction simplifies analysis. A level-ruled extension of a Petri net defines the physical interface of the system modeled by the Petri net. This is its input and output signals with the initial state of the outputs. The extension also includes the timing information and semantic abstraction implemented by the Boolean functions. The structure of the level-ruled extension is given in Definition 2.10.

Definition 2.10 (Level-ruled Extension). *A level-ruled extension of a Petri net is represented by the six-tuple $E = (W, L, \nu_o, \mathsf{Eft}, \mathsf{Lft}, \mathsf{Lsat})$ that defines its signal set, mapping, initial state, firing bounds, and level information.*

The level-ruled extension can be divided into two distinct groupings affecting different aspects of the Petri net: the first three members of the tuple correspond to the physical interface of the Petri net and provide a link to actual wires in the timed circuit; and the second three members of the tuple correspond to the rules that define the timing behavior and simplify the structure of the Petri net. Although the level-ruled extension appears large and cumbersome, its ability to amend the Petri net to timed circuit specification is elegant and simple.

The first three members W, ν_o, and L of the six-tuple in Definition 2.10 make the physical connection between the Petri net and the wires or signals in the timed circuit. W is a finite set of signals in the timed circuit including those used in the Boolean functions for the syntactic abstraction. It defines the physical interface for the level-ruled Petri net model of the timed circuit. Recall that a Petri net is the four-tuple $N = (T, P, F, \mu_o)$ in Definition 2.1; thus, the labeling function $L : T \to (W \times \{+, -\} \cup T)$ is a mapping from transitions in N to real actions on members of the signal set W. Transitions that are mapped into $W \times \{+, -\}$ are either rising or falling. A signal w that moves from a low to a high state is *rising* as indicated by $w+$, and a signal that moves from a high to low state is *falling* as indicated by $w-$. If a transition from T does not affect the state of a signal, then L returns the transition itself rather than a member of the $W \times \{+, -\}$ set. Although firing this transition has no affect on the Boolean state of the signal set, it does affect the state of the Petri net by moving it to a new marking. The output signal set is computed from the structure of the level-ruled Petri net.

Definition 2.11 (Output Signals). *A signal $w \in W$ is an output signal if there exists a transition $t \in T$ such that $R(t) \neq \emptyset$ and either $L(t) = w+$ or $L(t) = w-$; the set all output signals is O.*

A signal is an output if it has rules and transitions defined to control its behavior.If a signal only has a defined transition, but not defined rules, then it is a input because it effectively cannot be controlled; thus, a level-ruled Petri net cannot affect the behavior of any of its inputs. The set of input signals is given as $W - O$. Note that a transition that is not defined on a signal is neither an input or an output. The initial Boolean state of the output signal set is given by $\nu_o \subseteq O$. For all output signals $w \in \nu_o$, w is high in the initial Boolean state. For all outputs $w \notin \nu_o$, w is low in the initial Boolean state. Because ν_o is defined over the output set, a level-ruled Petri net is a representation of a module. A module implements the behavior of members in its output set. The behavior of the input set must be defined by some other module in the system; thus, a timed circuit is a network of modules where each module defines a portion of the system behavior.

The second three members Eft, Lft, and Lsat of the eight-tuple in Definition 2.10 relate to the rules that define the timing behavior and implement the syntactic abstraction in the Petri net. A rule can have three different properties: an earliest firing timing, a latest firing time, and a Boolean function defined over the signals in the timed circuit. The symbol $\mathbb{Q}^+$ represents the set of nonnegative rational numbers. The *earliest firing time* $\mathsf{Eft} : R \to \mathbb{Q}^+$ is a function mapping rules to nonnegative rational numbers. For a rule $r = (p, t)$, $\mathsf{Eft}(r)$ is a minimum separation that must exist between the firing of t and the satisfaction of the enabling conditions for r. The *latest firing time* $\mathsf{Lft} : R \to \mathbb{Q}^+ \cup \{\infty\}$ is defined similarly, except that the symbol ∞ is included to represent an infinite latest firing time. All members of $\mathbb{Q}^+$ are by definition less than ∞. $\mathsf{Lft}(r)$ is a possible maximum separation between the firing of t and the satisfaction of the enabling conditions for r. If a transition has several rules that affect its behavior, then it is possible for some rules to exceed their latest firing time before the transition fires. Note that the earliest and latest firing times are multiplied by their greatest common denominator to convert them to integers for analysis. The Lsat member of the level-ruled extension implements the syntactic abstraction. It is the function $\mathsf{Lsat} : R \to (2^W \to \{\mathsf{true}, \mathsf{false}\})$. The function takes a rule and a Boolean state defined over the signals in the level-ruled Petri net, and it returns either true or false. The true value indicates that the level information is satisfied by the state; the false value indicates the opposite. Like the definition for ν_o, a Boolean state, $\nu \subseteq W$, is a subset of the signal set. The inclusion of a signal indicates a high Boolean state and the exclusion the opposite.

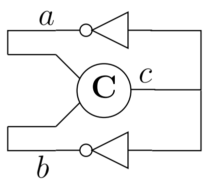
Consider the Petri net in Fig. 2.2. A level-ruled extension of this net can link it to a Muller C-element whose inputs are the inversion of its output [11]. The Muller C-element gate symbol is given in Fig. 2.3(a) with its truth table in Fig. 2.3(b). From the truth table, if the initial Boolean states of a, b, and c are low, then when both a and b transition high, c transitions high. Similarly, if a, b, and c are high, then when both a and b transition low, c transitions low. When the states of a and b are different, c holds its value. The output set for the level-ruled extension of Fig. 2.2 is $O = \{a, b, c\}$ because the Petri net defines all transition behaviors. There is an entry for each signal shown in Fig. 2.3(a). The initial Boolean state is given as $\nu_o = \emptyset$ because all signals are low. The set for the labeling function is

$$L = \left\{ \begin{array}{lll} (t_1, c-), & (t_2, a+), & (t_3, b+), \\ (t_4, c+), & (t_5, a-), & (t_6, b-) \end{array} \right\}$$

The labeling function is the link between transitions in the Petri net and actions on signals in the output set of the level-ruled extension; there is a rising and falling transition for each signal. Notice that with this labeling function, the behavior of c defined by the net resembles that of the Muller C-element, and the behaviors of a and b are just the inversion of c. The rule set is

$$R = \left\{ \begin{array}{llll} (p_1, t_2), & (p_2, t_3), & (p_3, t_4), & (p_4, t_4), \\ (p_5, t_5), & (p_6, t_6), & (p_8, t_1), & (p_7, t_1) \end{array} \right\}$$

This is each place-transition pair, (p, t), that is found in the flow relation in (2.1). To continue with the example, assume that the signals a and b are untimed, but



(a)

a	b	c
0	0	0
0	1	c
1	0	c
1	1	1

(b)

Fig. 2.3. The Muller C-element and its defined behavior. (a) The drawn gate symbol with a simple environment. (b) The truth table describing the gate's behavior.

the signal c has a rising delay of $[30, 50]$ and a falling delay of $[25, 45]$. From this, the earliest firing time relation is

$$\mathsf{Eft} = \left\{ \begin{array}{llll} ((p_1, t_2), 0), & ((p_2, t_3), 0), & ((p_3, t_4), 30), & ((p_4, t_4), 30), \\ ((p_5, t_5), 0), & ((p_6, t_6), 0), & ((p_8, t_1), 25), & ((p_7, t_1), 25) \end{array} \right\};$$

and the latest firing time relation is

$$\mathsf{Lft} = \left\{ \begin{array}{llll} ((p_1, t_2), \infty), & ((p_2, t_3), \infty), & ((p_3, t_4), 50), & ((p_4, t_4), 50), \\ ((p_5, t_5), \infty), & ((p_6, t_6), \infty), & ((p_8, t_1), 45), & ((p_7, t_1), 45) \end{array} \right\}.$$

The last member of the level-ruled extension is Lsat. It is defined such that for all rules $r \in R$ and all states $\nu \in 2^W$, $\mathsf{Lsat}(r)(\nu) = \mathsf{true}$; thus, it always returns true regardless of the rule or state. This is because there is no syntactic abstraction in Fig. 2.2—there are no Boolean functions.

Consider now the network of level-ruled Petri nets in Fig. 2.4. This network uses syntactic abstraction to capture the same behavior of a Muller C-element whose inputs are the inversion of its output as that of the Petri net in Fig. 2.2. Fig. 2.4(a) and Fig. 2.4(b) define the behavior of a and b, which are the inversion of c. These two nets are the environment definition for the Muller C-element. Fig. 2.4(c) defines the behavior of c, which is the Muller C-element. The level-ruled extension for each of these nets is depicted in their graphical representations. Consider Fig. 2.4(c) that defines the behavior of c. Its transitions are labeled $c+$ and $c-$ due to the labeling function in the level-ruled extension. Similarly, its rules are annotated with timing bounds and Boolean functions. For example, the rule for $c+$ includes the bound

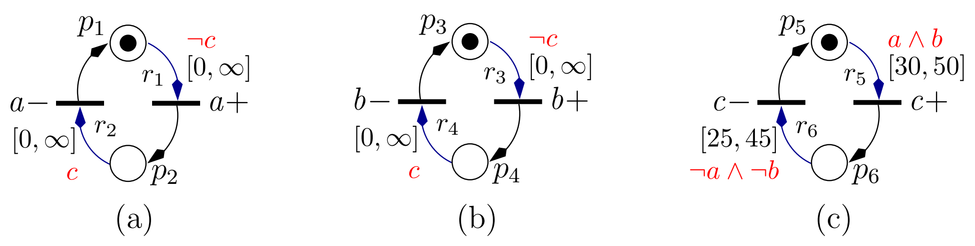


Fig. 2.4. A Muller C-element level-ruled Petri net model. (a) The specification for input a. (b) The specification for input b. (c) The specification for output c.

$[30, 50]$ for the earliest and latest firing times and the Boolean function $a \wedge b$. The Boolean function $a \wedge b$ is represented as

$$\mathsf{Lsat}(r_5) = \left\{ \begin{array}{llll} (\{\}, \mathsf{false}), & (\{c\}, \mathsf{false}), & (\{b\}, \mathsf{true}), & (\{b, c\}, \mathsf{true}), \\ (\{a\}, \mathsf{true}), & (\{a, c\}, \mathsf{true}), & (\{a, b\}, \mathsf{true}), & (\{a, b, c\}, \mathsf{true}) \end{array} \right\}.$$

Similar sets represent the Boolean functions in the other rules too. In this model of the Muller C-element and its environment, the syntactic abstraction facilitates breaking the system into modules and reduces the total number of places in the system. The number of reachable markings for the two systems, however, is identical.

A timed circuit is the parallel composition of a network of level-ruled Petri nets like that shown in Fig. 2.4. The composition results in a single net representing the behavior of the entire system. Recall that $E = (W, L, \nu_o, \mathsf{Eft}, \mathsf{Lft}, \mathsf{Lsat})$; this is used in the following definition.

Definition 2.12 (Parallel Composition). *The parallel composition of a network of level-ruled Petri nets $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is the single net $M = (N, E)$ where for $1 \leq i \leq n$ and $M_i = (N_i, E_i)$, N is the union over the constituent members of each N_i; and E is the union over the constituent members of each E_i except Lsat, which is a new function such that for all rules $r_i \in R_i$ and Boolean states $\nu \in 2^W$,*

$$\mathsf{Lsat}_i(r_i)(\nu') = \mathsf{true} \iff \mathsf{Lsat}(r_i)(\nu) = \mathsf{true},$$

where $\nu' = \nu \cap W_i$ is the Boolean state ν with all of the signals not in the component signal set W_i removed.

Syntactic abstraction must be specially treated in the parallel composition because the composition changes the signal set W over which the Boolean functions are defined.

A Boolean function in a given level-ruled Petri net is defined over signals on the interface of that net. When that net is moved into a parallel composition of a system, its signal set is absorbed into the system. The system has a new interface defined over every signal on the interface of all nets in the system. Any Boolean function belonging to a net that is part of the parallel composition must expand, if necessary, to accommodate the new interface for the system. The definition creates a new function for each Boolean function in every module such that any signal not on the interface of the module is treated as a *don't care*. Any Boolean function from any of the modules in the composition is Boolean equivalent to its counterpart in the parallel composition, only its counterpart is now defined over the larger signal set.

A parallel composition is not always correct because initial markings, Boolean states, as well as firing times and Boolean functions, for places, signals, or rules may be defined differently in different modules. A network creates a valid parallel composition if member modules consistently define shared objects.

Definition 2.13 (Consistent). *A pair of level-ruled Petri nets (M, M') is consistent if the following three conditions hold:*

1. *for all places $p \in (P \cap P')$, $p \in \mu_o \iff p \in \mu'_o$;*
2. *for all signals $w \in (O \cap O')$, $w \in \nu_o \iff w \in \nu'_o$; and*
3. *for all rules $r \in (R \cap R')$, $L(r) = L'(r)$, $\mathsf{Eft}(r) = \mathsf{Eft}'(r)$, $\mathsf{Lft}(r) = \mathsf{Lft}'(r)$, and $\mathsf{Lsat}(r) = \mathsf{Lsat}'(r)$.*

Two nets are consistent if they agree on the marking of shared places, the Boolean state of shared outputs, and identically define properties of shared rules.

Definition 2.14 (Valid Composition). *The parallel composition of a network of level-ruled Petri nets $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is valid if for all pairs (i, j) such that $1 \leq i, j \leq n$, (M_i, M_j) is consistent.*

The parallel composition of the network in Fig. 2.4 is valid. The initial marking, Boolean state, and rule set is consistently defined across all three nets as they do not share any transitions or places. An example of a network that shares transitions is presented in Section 2.3.3.

Definition 2.15 (Closed). *A parallel composition is closed if its output set O is equal to its signal set W.*

A closed composition completely defines the behavior of the network. The parallel composition of the network in Fig. 2.4 is closed. It completely defines all signals in its output set. Although the signals a and b are considered to be the environment for the circuit implementing c, their behavior is defined. This is because the behavior of c depends on a and b; thus, their behavior is required for any analysis of c. The rest of this presentation considers only closed systems for simplicity.

2.2.2 Semantics

The state of a level-ruled Petri net is defined by the three-tuple $(\mu, \nu, \mathcal{C})$: μ is the current marking of the net; ν is the current Boolean state of the signals in the net; and $\mathcal{C} : R \to \mathbb{R}^+ \cup \{\infty\}$ is a clock assignment function. The symbol $\mathbb{R}^+$ is the set of positive real numbers. The marking supports the Petri net, the Boolean state supports the syntactic abstraction, and the clock assignment supports time. Every rule in the level-ruled Petri net is associated with a clock. For a given rule r, $\mathcal{C}(r)$ returns the value of the clock on r. There are two operations on clocks: advance and reset. For some positive real number $d \in \mathbb{R}^+ \cup \{\infty\}$, $\mathcal{C} + d$ advances the clock for every $r \in R$ to the value $\mathcal{C}(r) + d$. For an unbounded delay $d = \infty$, however, $\mathcal{C} + d$ sets the clock for every $r \in R$ to $\mathcal{C}(r) = \infty$. For some subset of rules $\hat{R} \subseteq R$, $\left[\hat{R} \mapsto 0\right] \mathcal{C}$ resets the clock for every rule in $\hat{R}$ to zero and agrees with $\mathcal{C}$ for every rule in $R - \hat{R}$. The initial clock assignment for the system is defined such that the clock for every rule is zero and is denoted by $\mathcal{C}_o$.

The state of a level-ruled Petri net can change by firing a transition or by advancing time. To fire a transition t from the state $(\mu, \nu, \mathcal{C})$, it must be marking satisfied in μ. In addition to this, it must satisfy conditions in the Boolean state ν and the clock assignment $\mathcal{C}$ too.

Definition 2.16 (Marking Satisfied Rule Set). *A set of rules R' is satisfied by the marking μ if $P' \subseteq \mu$ where $P' = \{p \in P \mid (p, t) \in R'\}$ is the set of places for the rules in R'; the operator $\vdash$ denotes this property; $\mu \vdash R'$ indicates that the property* *holds and $\mu \nvdash R'$ the opposite.*

Definition 2.17 (Marking Satisfied Transition). *A transition t is marking satisfied by μ if $\mu \vdash R(t)$.*

Definition 2.2 on Petri net semantics is identical to Definition 2.17. It is expressed in terms of rules to make the notation consistent with level-ruled Petri nets.

A set of rules is *level satisfied* by a Boolean state if the Boolean function for each rule in its rule set returns true on this state.

Definition 2.18 (Level Satisfied Rule Set). *The rule set R' is level satisfied by the Boolean state ν if for all $r \in R'$, $\mathsf{Lsat}(r)(\nu) = \mathsf{true}$; $\nu \vdash R'$ indicates that the property holds and $\nu \nvdash R'$ the opposite.*

Definition 2.19 (Level Satisfied Transition). *A transition t is level satisfied by ν if $\nu \vdash R(t)$.*

For the network in Fig. 2.4, the transitions $a+$ and $b+$ are level satisfied by the initial state $\nu_o = \emptyset$ because $\mathsf{Lsat}(r_1)(\emptyset) = \mathsf{true}$ and $\mathsf{Lsat}(r_3)(\emptyset) = \mathsf{true}$. Consider the fragment shown in Fig. 2.5. If the current Boolean state is $\nu = \{a, b\}$, then the transition t_1 is not level satisfied because $\mathsf{Lsat}(r_1)(\{a, b\}) = \mathsf{true}$ while $\mathsf{Lsat}(r_2)(\{a, b\}) = \mathsf{false}$. If the current Boolean state is $\nu = \{a, b, c\}$, however, then t_1 is level satisfied because $\mathsf{Lsat}(r_2)(\{a, b, c\})$ is now true.

The time semantics define the earliest time at which it is possible to perceive a change in the Boolean state of a wire after it is enabled to transition. It also defines a latest possible time where the state is ensured to have changed. This maps the analogue phenomenon of a signal moving between its bistable states into

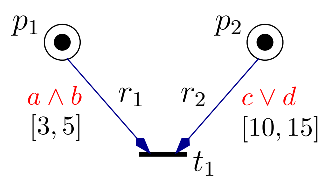


Fig. 2.5. A net fragment showing a transition with two rules.

the level-ruled Petri net model. The time spent between the bistable states is undefined; thus, until an enabled transition on a signal fires, the state of the signal is unknown.

A set of rules is *time satisfied* by a clock assignment function if two conditions are met: first, the clock for each rule in its rule set is above its earliest firing time; and second, if its rule set is not empty, then there exists a clock for a rule in its rule set, that is below its latest firing time. These two conditions are formalized in Definition 2.20.

Definition 2.20 (Time Satisfied Rule Set). *The rule set R' is time satisfied by the clock assignment function $\mathcal{C}$ if for all $r \in R'$, $\mathcal{C}(r) \geq \mathsf{Eft}(r)$; and if $R' \neq \emptyset$, then there exists a rule $r' \in R'$ such that $\mathcal{C}(r') \leq \mathsf{Lft}(r')$; $\mathcal{C} \vdash R'$ indicates that the property holds and $\mathcal{C} \nvdash R'$ the opposite.*

Definition 2.21 (Time Satisfied Transition). *A transition t is time satisfied by $\mathcal{C}$ if $\mathcal{C} \vdash R(t)$.*

Consider again the fragment shown in Fig. 2.5. The transition t_1 is time satisfied in the shown marking and Boolean state, $\{a, c\}$, if $\mathcal{C}(r_1) = 8$ and $\mathcal{C}(r_2) = 14$ as each is above its earliest firing time and $\mathcal{C}(r_2)$ is below its latest firing time. If $\mathcal{C}(r_1) = 10$ and $\mathcal{C}(r_2) = 16$, however, then t_1 is not time satisfied because the clocks for r_1 and r_2 have exceeded the latest firing time of their respective rules.

Only enabled transitions can fire in a state of the level-ruled Petri net. For a transition to be enabled, it must meet properties in all members of the state tuple.

Definition 2.22 (Enabled Transition). *The transition t is enabled in the state $(\mu, \nu, \mathcal{C})$ if $\mu \vdash R(t)$, $\nu \vdash R(t)$, and $\mathcal{C} \vdash R(t)$; $(\mu, \nu, \mathcal{C}) \vdash R(t)$ indicates that the property holds and $(\mu, \nu, \mathcal{C}) \nvdash R(t)$ the opposite.*

Consider a state $(\mu, \nu, \mathcal{C})$ of the network of level-ruled Petri nets in Fig. 2.4. Suppose that μ is the one shown, $\nu = \emptyset$ as all signals are currently in a low Boolean state, and the clock assignment function is defined such that $\mathcal{C}(r_1) = 5$ and $\mathcal{C}(r_3) = 5$. The transitions $a+$ and $b+$ are enabled in this state. Each is marking satisfied as

$\mu \vdash R(a+)$ and $\mu \vdash R(b+)$; each is level satisfied as $\neg c$ is true in ν; and each is time satisfied by $\mathcal{C}$ as 5 is greater than the earliest firing time defined on r_1 and r_3. The transition $c+$, however, is not enabled in this state. Although it is marking satisfied by μ, it is not level satisfied by the state as $a \wedge b$ is currently false.

Firing an enabled transition updates each member of the state tuple to reflect the new state of the system. The new marking is given by Definition 2.3. The Boolean state is updated by adding to or removing from the current Boolean state the output associated with the transition if one exists.

Definition 2.23 (Boolean State Update). *The Boolean state created from firing an enabled transition t in $(\mu, \nu, \mathcal{C})$ is computed as*

$$\nu' = \begin{cases} \nu \cup \{u\} & \textit{if}\, L(t) = u+, \\ \nu - \{u\} & \textit{if}\, L(t) = u-, \ \textit{and} \\ \nu & \textit{otherwise}. \end{cases}$$

The first two cases on the update handle the low to high and high to low transitions of an output signal. The last case is for transitions that are not mapped to actions on outputs. These transitions affect only the marking and do not affect the Boolean state of the system. If the enabled transition $a+$ is fired in Fig. 2.4(a) from the shown marking with $\nu = \emptyset$, then from Definition 2.23, $\nu' = \{a\}$.

The update on the clock function defines the rules for which clocks are to be reset and then resets those clocks.

Definition 2.24 (Reset Rules). *The set of reset rules $\hat{R}$ in $(\mu, \nu, \mathcal{C})$ when firing the enabled transition t is computed as $r \in \hat{R}$ if $\mathsf{Lsat}(r)(\nu') = \mathsf{true}$ in the updated Boolean state ν'; and either $r \in \{(p', t') \in R \mid p' \in t\bullet\}$, or $r \in \{(p', t') \in R \mid p' \in \mu\}$ and $\mathsf{Lsat}(r)(\nu) = \mathsf{false}$.*

The first requirement on members of the reset rule set is that each rule in the set be level satisfied by the new Boolean state created by firing the enabled transition. The next requirement on the rules in the set can be satisfied in two different ways: first, the firing of the transition adds the rule's place to the marking; or second, the rule's place is already in the marking but firing the transition causes it to become level satisfied where it is currently not level satisfied. Consider the net fragment in Fig. 2.4(a). Firing $a+$ from the shown marking in the initial Boolean state creates an empty set of reset rules. This is because the rules on $a-$ and $c+$ are not level satisfied by $\nu = \{a\}$ even though p_2 is newly added to the marking. The rule for $b+$ is not reset either as its place is not newly added to the marking and it is already level satisfied by the initial state. Firing $b+$ next, however, creates a set of reset rules $\hat{R} = \{r_4\}$ as the new Boolean state $\nu = \{a, b\}$ causes the rule r_4 on $c+$ to become level satisfied.

Definition 2.25 (Clock Assignment Update). *The clock assignment created from firing an enabled transition t in $(\mu, \nu, \mathcal{C})$ is computed as $\mathcal{C}' = \left[\hat{R} \mapsto 0\right] \mathcal{C}$ where $\hat{R}$ is the set of reset rules.*

The clock assignment updates the final member of the state tuple by resetting clocks on rules in the reset set; thus, the final state of the network in Fig. 2.4 after firing $a+$ from $(\mu, \nu, \mathcal{C})$ where μ is the one shown, $\nu = \emptyset$, and $\mathcal{C}$ is such that $\mathcal{C}(r_1) = 5$ and $\mathcal{C}(r_3) = 5$, is $(\mu', \nu', \mathcal{C}')$ where $\mu' = \{p_2, p_3, p_5\}$, $\nu' = \{a\}$, and $\mathcal{C}' = \mathcal{C}$. If the enabled transition $b+$ is now fired from the new state, then the updated marking is $\{p_2, p_4, p_5\}$, the updated Boolean state is $\{a, b\}$, and the updated clock assignment agrees with the new clock assignment except that now the clock value for r_5 is zero as it is reset when the rule became level satisfied due to the firing of $b+$.

The state of the level ruled Petri net can change not only by firing an enabled transition, but also by advancing time, which is to fire a delay. Unlike firing a transition, firing a delay only affects the clock assignment function in the state tuple.

Definition 2.26 (Delay Firing). *The new state created from firing the delay $d \in \mathbb{R}^+ \cup \{\infty\}$ in $(\mu, \nu, \mathcal{C})$ is given as $(\mu, \nu, \mathcal{C}')$ where $\mathcal{C}' = \mathcal{C} + d$.*

As shown, firing a delay is an advancement of time in the clock assignment function. There is, however, a restriction on the size of the delay that can be fired. This is described as an enabling condition; thus, only enabled delays can fire from a state.

Definition 2.27 (Maximum Delay). *The maximum delay* $d \in \mathbb{R}^{+} \cup \{\infty\}$ *that can fire in the state* $(\mu, \nu, \mathcal{C})$ *given the set* $X = \{t \in T \mid R(t) \neq \emptyset \wedge (\mu, \nu) \vdash R(t)\}$ *of marking and level satisfied transitions in the state with nonempty rule sets is*

$$d = \begin{cases} \infty & \text{if } X = \emptyset \text{ and} \\ \min_{t \in X}\left(\max_{r \in R(t)}\left(\mathsf{Lft}(r) - \mathcal{C}(r)\right)\right) & \text{otherwise;} \end{cases}$$

The calculation only considers transitions that are marking and level satisfied in the state because these are the transitions that are either currently time satisfied and thus, enabled, or will become such by advancing time. If none of these transitions, have defined rule sets as indicated by $X = \emptyset$, then the maximum delay is infinite. Any enabled transition in the state can fire at any time in this scenario. If X is nonempty, however, then the maximum delay is computed by looking at the rule sets for each of the transitions in X. Consider the inner part of the computation in Definition 2.27 that is given as $\max_{r \in R(t)}\left(\mathsf{Lft}(r) - \mathcal{C}(r)\right)$. For each rule in a transition's rule set, it finds the maximum amount of time that can advance while preserving a single rule in the set that is time satisfied after the advancement. The maximum delay transition is the minimum of these delays over all of the transitions that are marking and level satisfied by the state. This means that a transition either remains nonenabled, becomes enabled, or is enabled already and remains enabled after the delay transition. Consider the state $(\mu, \nu, \mathcal{C})$ of the net in Fig. 2.4 where $\mu = \{p_2, p_4, p_5\}$, $\nu = \{a, b\}$, and $\mathcal{C}$ is such that $\mathcal{C}(r_5) = 0$. Transition $c+$ is the only transition that is marking satisfied by μ and level satisfied by ν; thus, the maximum delay computation is $d = 50$. Now consider the net fragments in Fig. 2.6. The marking is $\mu = \{p_1, p_2, p_3\}$; the Boolean state is $\nu = \{a, b, c\}$; and the clock assignment function is such that $\mathcal{C}(r_1) = \mathcal{C}(r_2) = \mathcal{C}(r_3) = 5$. Transitions t_1 and t_2

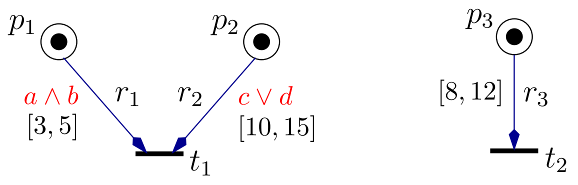


Fig. 2.6. Net fragments illustrating the maximum delay definition.

are marking satisfied by μ and are level satisfied by ν, but they are not enabled in the state $(\mu, \nu, \mathcal{C})$ because they are not time satisfied by $\mathcal{C}$. The maximum delay in this state is $d = 7$. Although transition t_1 can accept $d = 10$, a delay of this size would cause transition t_2 to never be time satisfied. To illustrate this, suppose that the delay of 10 is fired in this state. The clock assignment in the new state for r_3 is now $\mathcal{C}'(r_3) = 15$. Transition t_2 can never be time satisfied from this state by Definition 2.21 as $\mathcal{C}'(r_3) > \mathsf{Lft}(r_3)$, and r_3 is its only rule; thus, t_2 never becomes enabled and fires after the time advancement. The outer part of the maximum delay computation is a minimum over all transitions that are marking and level satisfied by the state to not miss behaviors by allowing time to advance beyond the point where transitions can fire.

Definition 2.28 (Enabled Delay). *The delay $d \in \mathbb{R}^+ \cup \{\infty\}$ is enabled in a state if d is equal to or less than the maximum delay allowed by that state; this is indicated for some state s and delay d by $s \vdash d$; $s \nvdash d$ indicates the property does not hold.*

Any rule that is less than or equal to the maximum delay in a given state is an enabled delay transition by Definition 2.28. This prevents a transition from loosing its enabling or not becoming enabled at all through the advancement of time; thus, behaviors in the model are not lost through the advancement of time.

It is convenient to define actions on delay-transition pairs as a level-ruled Petri net is often first updated by firing a delay, and then updated by firing a transition. The following definitions are presented to support this notion.

Definition 2.29 (Enabled Delay-transition Pair). *The delay-transition pair (d, t) is enabled in the state $(\mu, \nu, \mathcal{C})$ if the delay d is enabled in $(\mu, \nu, \mathcal{C})$ and firing it from $(\mu, \nu, \mathcal{C})$ leads to the state $(\mu, \nu, \mathcal{C}')$ where the transition t is enabled; this is indicated by $(\mu, \nu, \mathcal{C}) \vdash (d, R(t))$; $(\mu, \nu, \mathcal{C}) \nvdash (d, R(t))$ indicates that the property does not hold.*

For the state $(\mu, \nu, \mathcal{C})$ of the network in Fig. 2.4 where $\mu = \{p_2, p_4, p_5\}$, $\nu = \{a, b\}$, and $\mathcal{C}$ is such that $\mathcal{C}(r_5) = 0$, the delay-transition pair $(35, c+)$ is enabled. This is because the delay 35 is enabled in the state and firing the delay 35 leads to a new state where $c+$ is enabled to fire. It is important to note that a delay does not always have to be nonzero. Consider now the state of the same network where the μ is the one shown, $\nu = \emptyset$, and $\mathcal{C}$ is such that the clocks for all rules are at zero. The set of enabled transitions in this state is $\{a+, b+\}$; thus, the delay-transition pair $(0, a+)$ is enabled as is the pair $(0, b+)$. In reality, there are an infinite number of enabled delay-transition pairs in this state as the transitions $a+$ and $b+$ are untimed.

Definition 2.30 (State Update). *The state created by firing the enabled delay-transition pair (d, t) in $(\mu, \nu, \mathcal{C})$ is given as $(\mu', \nu', \mathcal{C}')$ where μ' and ν' are the marking and Boolean state updated with the firing of t, and $\mathcal{C}'$ is the clock assignment that is first updated with the firing of d followed by t.*

Suppose that s is the state $(\mu, \nu, \mathcal{C})$ and that s' is the state $(\mu', \nu', \mathcal{C}')$. The notation $s\,[(d, t)\rangle\, s'$ is used to indicate that firing the enabled delay-transition pair (d, t) in s leads the system to s'. Forcing the delay to fire before the transition lets time advance so that transitions can become time satisfied and enabled to fire.

A *firing sequence* in a level-ruled Petri net is a pair of delay and transition vectors $(\mathbf{d}, \mathbf{t})$ of equal length such that firing each delay-transition pair (d_i, t_i) leads to a state tuple where the next one can fire. This is expressed in Definition 2.31.

Definition 2.31 (Firing Sequence). *A pair of delay and transition vectors $(\mathbf{d}, \mathbf{t})$ where $\mathbf{d} = (d_1, d_2, \ldots, d_{n-1}, d_n)$ and $\mathbf{t} = (t_1, t_2, \ldots, t_{n-1}, t_n)$ is a firing sequence if there exists a vector of states $\mathbf{s} = (s_0, s_1, s_2, \ldots, s_{n-1}, s_n)$ such that for each $s_i = (\mu_i, \nu_i, \mathcal{C}_i)$, the following holds: (d_i, t_i) is enabled in s_{i-1} and $s_{i-1}\,[(d_i, t_i)\rangle\, s_i$ for $1 \leq i \leq n$; the function $\mathcal{P}(s)$ returns the set of all possible firing sequences that start from the state $s = (\mu, \nu, \mathcal{C})$.*

The pair $(5, 150, 30)$ and $(b+, a+, c+)$ is a firing sequence for the network in Fig. 2.4 that satisfies Definition 2.31. Delay-transition pairs $(5, b+)$ and $(150, a+)$ are enabled in the shown initial state of the network regardless of the clock assignment. States exist such that firing them in any order enables $(30, c+)$. The pair $(5, 150, 20)$ and $(b+, a+, c+)$, however, is not a firing sequence because there does not exist an s that satisfies Definition 2.4; there is no state vector that enables $(20, c+)$ on this firing sequence.

A firing sequence is *prefix-closed* if any prefix of the firing sequence is a firing sequence too. A firing sequence is prefix-closed by definition. A firing sequence is *monotonic* if time can only advance in the forward direction. A firing sequence is also monotonic by definition. The *empty* firing sequence is represented by the pair (ϵ, ϵ). It is a firing sequence with no delay-transition pairs.

The notion of reachable states in a level-ruled Petri net can now be approached. One state is *reachable* from another state if there exists a firing sequence that takes the net from the one state to the other. This is expressed in Definition 2.32.

Definition 2.32 (Reachable). *The state s_n is reachable from s_o if there exists a firing sequence that starts in s_o and ends in s_n.*

The notation $s_o\,[(\mathbf{d}, \mathbf{t})\rangle\, s_n$ indicates that s_n is reachable from s_o on the firing sequence $(\mathbf{d}, \mathbf{t})$.

The set of *reachable states* defines the behavior of the level-ruled Petri net. This set is denoted by $[s\rangle$. Definition 2.6 formalizes this set.

Definition 2.33 (Reachable State Set). *The reachable states from an initial state s is the set of states $[s\rangle \subseteq 2^P \times 2^W \times \{\mathcal{C} \mid \mathcal{C} : R \rightarrow \mathbb{R}^+ \cup \{\infty\}\}$ where $s' \in [s\rangle$ if either $s' = s$ or there exists a firing sequence $(\mathbf{d}, \mathbf{t})$ such that $s\,[(\mathbf{d}, \mathbf{t})\rangle\, s'$.*

The reachable state space of a level-ruled Petri net is its reachable state set $[s_o\rangle$ derived from its initial state $s_o = (\mu_o, \nu_o, \mathcal{C}_o)$, where $\mathcal{C}_o$ is the clock assignment function such that all clocks are zero for all rules. The function $\mathcal{P}(s_o)$ returns the set of all possible firing sequences that start from the initial state.

2.3 Examples

The level-ruled Petri net can be used to specify a variety of timed systems. The goal of this section is to present select examples that demonstrate various properties of the level-ruled Petri net. Section 2.3.1 presents a model of the IBM elastic synchronous pipeline. This is a purely synchronous design and shows how the level-ruled Petri net can be applied in a synchronous domain. Section 2.3.2 is another synchronous design from IBM, but this design uses delayed-reset domino gates instead of static gates as seen in Section 2.3.1. This example is unique in that the level-ruled Petri net is used to model a system with many different clock domains. The aggressive nature of these circuits yields two-sided timing constraints that are critical for functionality.

The final example in this section, the STARI FIFO, is different from the previous two in that it does not include any type of syntactic abstraction. The examples in Section 2.3.1 and Section 2.3.2 both use Boolean functions to simplify the structure of the composed system. Section 2.3.3 describes the STARI FIFO at the module level without using any type of syntactic abstraction. The STARI FIFO is an asynchronous FIFO. It is included here to demonstrate how a network of level-ruled Petri nets not using syntactic abstraction is composed to create a single net model of the system.

2.3.1 Elastic Synchronous Pipelines

This is an example of the level-ruled Petri net applied to the design shown in Fig. 2.7. This is the IBM elastic pipeline from [12]. The pipeline is completely analyzed in Section 6.3.1. The goal of the design is to reduce power by fine grain clock gating. The presentation here is to showcase the ability of the level-ruled Petri net to specify real designs. There are four unique gates in this pipeline: an inverter that is represented by a small bubble on an input, an *AND* gate, a latch, and the environments for *clk* and *stall*. The level-ruled Petri net model for the inverter is given in Fig. 2.4(a) with the model for the *AND* gate given in Fig. 2.1(b). The latch and environment entities are shown in Fig. 2.8. The latch in Fig. 2.8(a) is transparent on the positive phase of the clock input and opaque on

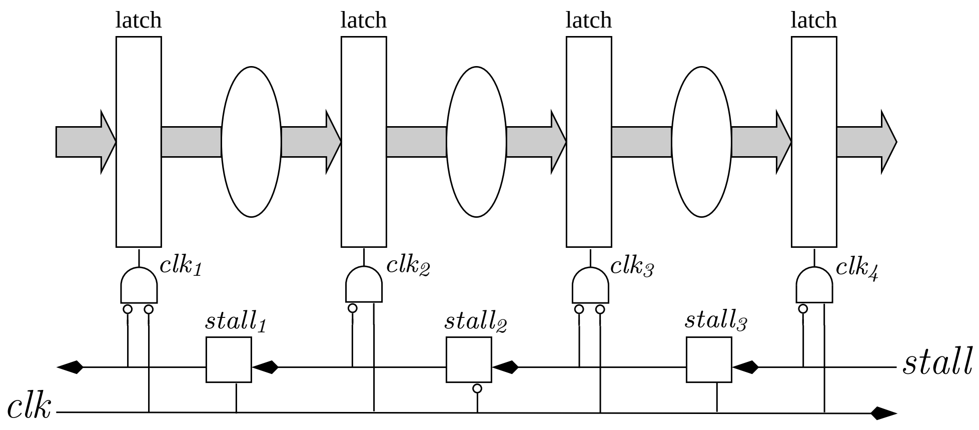


Fig. 2.7. A gate schematic of the IBM elastic synchronous pipeline.

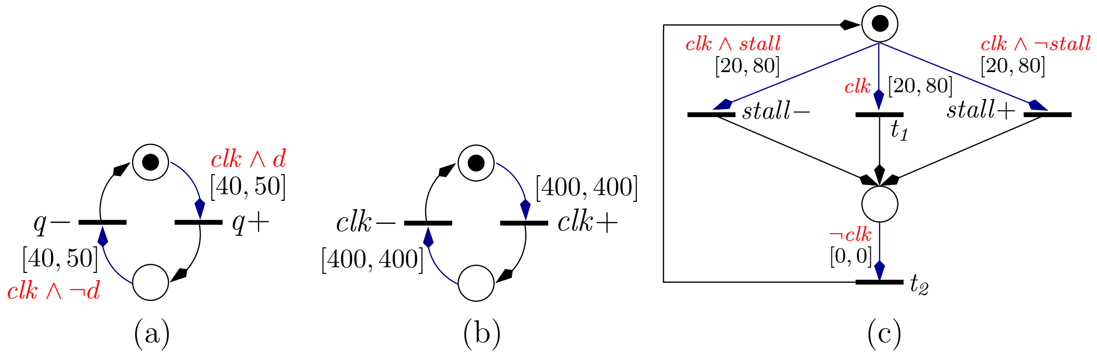


Fig. 2.8. The latch and environment models for the elastic pipeline. (a) The level-ruled Petri net model of an edge-triggered latch. (b) The level-ruled Petri net model for the clock input. (c) The level-ruled Petri net model of a random environment that captures all behaviors of the *stall* signal.

the negative phase. The environment in Fig. 2.8(b) is the global clock input. This model simply toggles the *clk* signal with a regular period. The final environment model in Fig. 2.8(c) produces firing sequences that represent the system being infinitely stalled, never stalled, and every stall behavior between the two extremes.

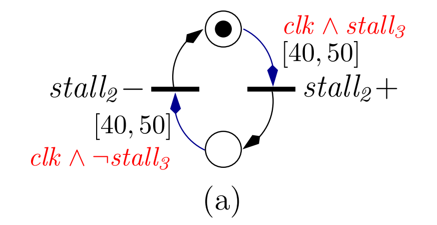
The structure of this net forces a random choice based on the state of the *stall* signal in the marking shown. If *stall* is high, the net can either fire the *stall−* transition, or it can do nothing by firing t_1. Similarly, if *stall* is low, then the net can either fire it high, or do nothing too. The random *stall* input enables an analysis algorithm to check correctness under all possible input conditions. The effect of this is that a correct implementation is correct under all input scenarios and timing conditions allowed by the environment and circuit model. This result is more important than one obtained through the simulation of inputs on corner cases. Another important property of the *stall* environment in Fig. 2.8(c) is that it does not generate multiple *stall* transitions in a given clock cycle. It either transitions *stall* high or low; or it does nothing. This type of environment is optimistic and assumes an input that is free of hazards, although it is possible to model an environment with hazards if needed.

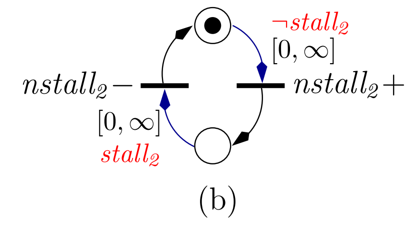
Using the models for the four basic gates, a complete model of the elastic pipeline in Fig. 2.7 can be constructed. This is done by first assigning unique names to the outputs of each gate in Fig. 2.7. The level-ruled Petri net model for each gate is then replicated, and the output names are mapped into each level-ruled Petri net according to the circuit connectivity. This completes the specification. The result of the replication and mapping for the second stage is shown in Fig. 2.9. There are four models in this figure. Fig. 2.9(a) is the latch to generate $stall_2$ based on its input $stall_3$. Fig. 2.9(b) is the invert of the $stall_2$ input. It generates $nstall_2$. Fig. 2.9(c) creates the actual clock input to the data latches. It is an *AND* gate. Its output is clk_2. Finally, Fig. 2.9(d) is a data latch to model a single bit of the data path.

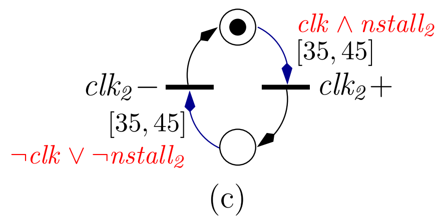
This elastic synchronous pipeline shows the level-ruled Petri net modeling a completely synchronous system at the gate level. In this application, the model can be analyzed to check for hazard freedom in the clock gating circuits. Without the syntactic abstraction, the structural complexity of this model is greatly increased.

2.3.2 Delayed-reset Domino Gates

The next example is a delayed-reset domino gate from the IBM gigahertz unit test site [4]. The test site is an experiment to build a gigahertz processor using a 600 nanometer fabrication process—an extremely high frequency for this process technology. To obtain this type of frequency, the processor is implemented using delayed-reset domino gates like that shown in Fig. 2.10. Belluomini describes this gate and presents a similar model in [13, 10]. The gate computes the function







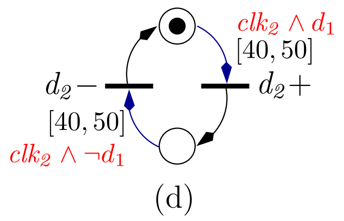


Fig. 2.9. The replication and mapping for the specification of the second stage. (a) The latch to generate the $stall_2$ signal. (b) The invert gate that takes the $stall_2$ signal and generates its inversion $nstall_2$. (c) The AND gate to generate the clk_2 input to the data latch. (d) A data latch to model a single bit of the data path.

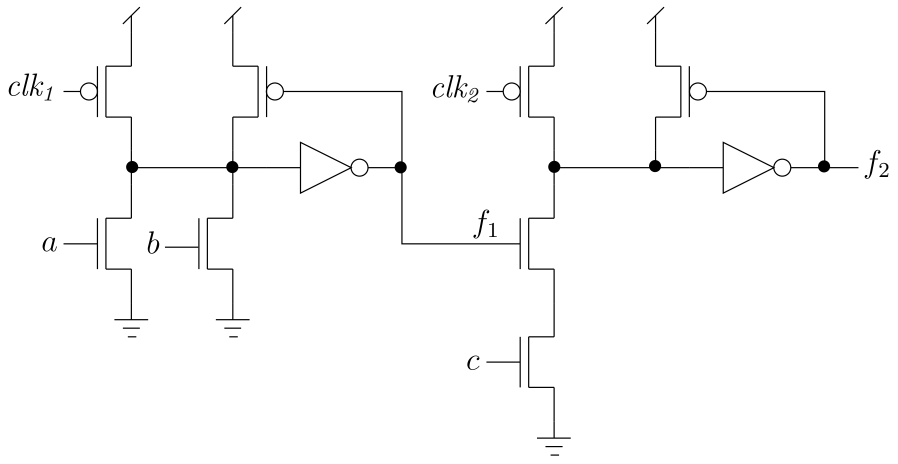


Fig. 2.10. A two-stage delayed-reset domino gate to compute $(a \vee b) \wedge c$.

$(a \vee b) \wedge c$ using two distinct stages. The first stage computes $f_1 = a \vee b$. The result is forwarded to the second stage that computes $f_2 = f_1 \wedge c$. Each stage of the gate is controlled by its own unique clock: clk_1 for stage one and clk_2 for stage two. Each stage precharges in the low phase of its clock. This drives its output to a low state. Each stage evaluates on the high phase of its clock. The delayed-reset domino gate is rather aggressive in design because it lacks a transistor at the bottom of each n-stack to turn off the ground path when the stage enters its precharge phase; thus, the timing of *clk1* and *clk2*, as well as other clocks in the system, must be such that inputs to any stage of the design are low before the stage enters its precharge phase. This prevents a stage from ever having a conducting path from power directly to ground that could potentially destroy it, or at a minimum, waste power.

The level-ruled Petri net model of the gate in Fig. 2.10 is given in Fig. 2.11. Fig. 2.11(a) is stage one of the gate. It computes $f_1 = a \vee b$ when *clk1* is high. If *clk1* is low, however, it simply sets f_1 to a low state. Fig. 2.11(b) is stage two of the gate. It computes $f_2 = f_1 \wedge c$ using the result of f_1 from stage one when *clk2* is high. Like stage one, it sets the function to a low state when *clk2* is low. A portion of a potential environment for the gate is shown is in Fig. 2.12. Fig. 2.12(a) is a global clock used to synchronize various local clocks. Fig. 2.12(b) is the local clock *clk1*. It operates the first stage that computes $f_1 = a \vee b$. Fig. 2.12(c) is an environment to generate input a. Like the environment shown in Fig. 2.8(c), this environment randomly selects to raise the a input or to leave it in its low state. The environments for inputs b and c are similar, only the input c has a larger delay so that it does not arrive at stage two until stage two is in its evaluate phase of *clk2*.

The delayed-reset domino gates shows the application of the level-ruled Petri nets to nonstandard designs. These gates are aggressive, and require special analysis to not only synthesize, but to demonstrate correctness. The level-ruled Petri net retains enough behaviors from the actual system to make it suitable for this type of analysis.

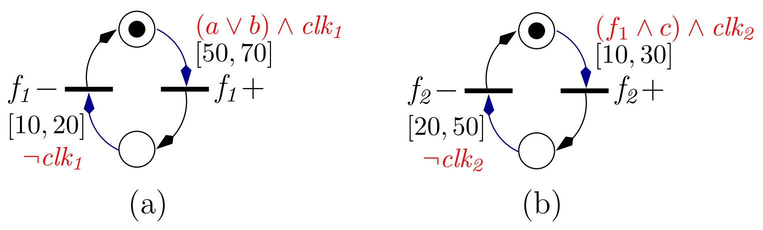


Fig. 2.11. The net models for the delayed-reset domino gate. (a) The net to compute $f_1 = a \vee b$. (b) The net to compute $f_2 = f_1 \wedge c$.

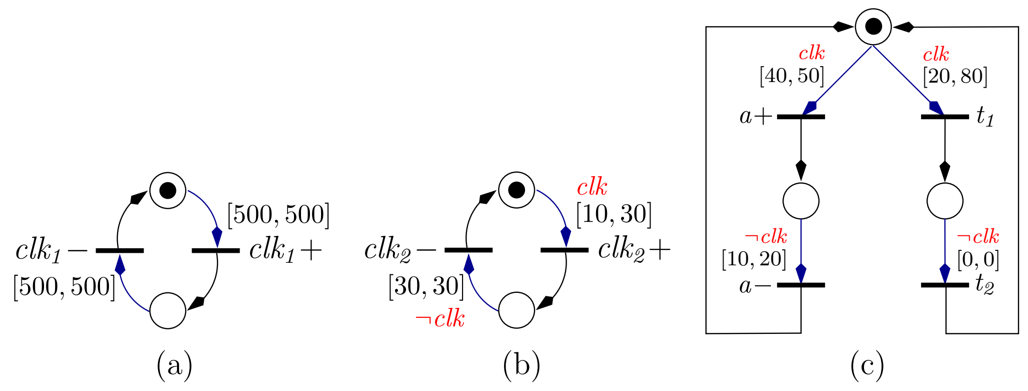


Fig. 2.12. Three environment models for the delayed-reset domino gate. (a) The model for the global clock. (b) The model for the first offset clock clk_1. (c) The model for the input a.

2.3.3 STARI FIFO

The next example is the STARI FIFO, a version of which is used by Sun Microsystems in a commercial application [14, 15]. Consider the block diagram of a STARI FIFO with two stages in Fig. 2.13. The STARI FIFO enables communication between two circuits that are operating at the same clock frequency but are out-of-phase due to clock skew [16, 17]. Clock skew can cause it to appear that one of the circuits operates faster than the other. The STARI protocol puts more data in the FIFO when the transmitter works faster than the receiver and supplies data from the FIFO to the receiver when the receiver works faster. The STARI FIFO is a common timed circuit benchmark, since its correctness depends on timing assumptions.

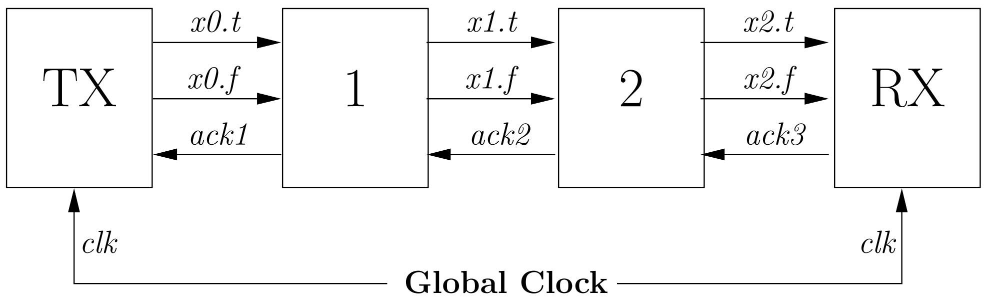


Fig. 2.13. The block diagram of a dual-rail STARI FIFO with two stages.

The functionality of the STARI FIFO can be described as follows. At the beginning of each clock period, one data item is inserted into the FIFO by the transmitter (TX) by setting either *x0.t* or *x0.f* high. At the same time, one data item is removed by the receiver (RX) by setting *ack3* low. Data is then allowed to propagate asynchronously down the FIFO queue. When *clk* goes low, the TX removes the input data item by resetting *x0.t* or *x0.f* and the RX removes the acknowledgment by setting *ack3* high.

The simplest level-ruled Petri net is the one shown in Fig. 2.14(a) that models the global clock. This model toggles the clock with a fixed period of 12 time units. Note that the majority of the places are not drawn to simplify the figures in this example. The transitions are replaced with their labels from the labeling function too. The level-ruled Petri net for the TX transmitter module is shown in Fig. 2.14(b). The drawn places in this figure represent conflict places. The *clk* signal is an input, and the *x0.t* and *x0.f* signals are outputs. When the net for the TX block sees the transition *clk*+, it randomly produces either $x0.t+$ or $x0.f+$ in the specified timing bound. This is the dual-rail encoded data item that is sent to the first stage. The 0 to 1 time delay is used to model the clock skew. After transmitting the data item, the TX module waits for the *clk*+ transition. It then resets either $x0.t+$ or $x0.f+$ depending on what it transmitted. The level-ruled Petri net for the RX receiver module in shown in Fig. 2.14(c). The *clk* signal is an input and the *ack3* signal is an output. This module waits for *clk*+ to transition and then lowers *ack3* to indicate that it has received the data item. After the *clk*+ transition, it raises its *ack3* line to request a new data item. Again, the 0 to 1 time delay is used to model clock skew.

The level-ruled Petri net for the first stage of the STARI FIFO is presented in Fig. 2.15(a). This stage does not currently hold a data item, and its *ack1* output is high. The model for the stage is divided into two nets. The lower net waits for either the $x0.t+$ or $x0.f+$ transition to indicate it has a valid data item. It also waits for *ack2* to go high to indicate that the next stage is empty and ready to receive a data item. It then fires its $x1.t+$ or $x1.f+$ transition to pass the data

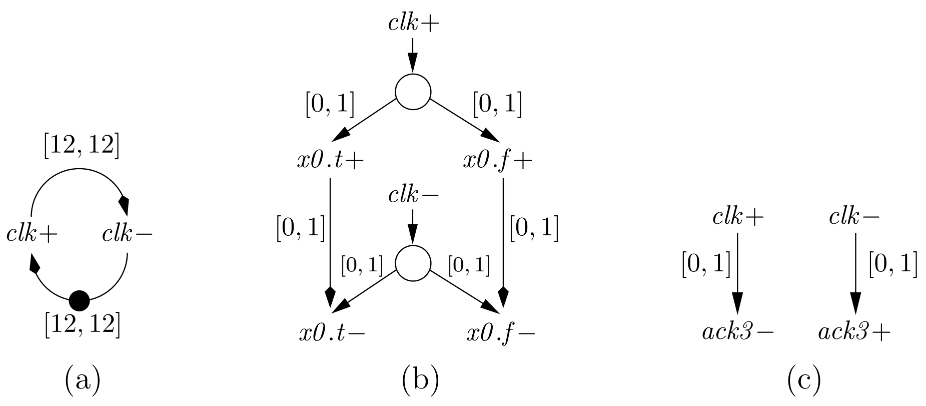


Fig. 2.14. The environment specification for the STARI FIFO. (a) The level-ruled Petri net for the global clock module. (b) The level-ruled Petri net for the TX transmitter module. (c) The level-ruled Petri net for the RX receiver module.

item to the next stage. After firing one of these two transitions, it fires *ack1−* to indicate to the TX module that it has successfully received the data item. At this point, it waits for either *x0.t−* or *x0.f−* and *ack2−* from the next stage to indicate that it has accepted the data item. It then resets by firing either *x1.t−* or *x1.f−* depending on the data item it transmitted. After firing one of these transitions, it fires its *ack1+* transition to request a new data item. The level-ruled Petri net model for the second stage is shown in Fig. 2.15(b) is similar to the first stage, only it starts in a different state because it is initialized with a data item.

If each component is examined separately, then some contain unconstrained input behaviors. Consider again the net shown in Fig. 2.14(a). In the initial marking, only the delay-transition pair (12,*clk+*) is enabled. Firing it leads to a state where the delay-transition pair (12,*clk−*) is enabled. Firing this leads again to the enabling of the first delay-transition pair (12,*clk+*). The behavior of the *clk* signal is completely defined in this module and is the only output signal. Consider now the level-ruled Petri net shown in Fig. 2.14(b). The transitions *clk+* and *clk−* are the only marking and level satisfied transitions in this net. They can fire with any delay. They are unconstrained inputs to this module. Suppose that *clk+* fires on some delay. After it fires, either *x0.t+* or *x0.f+* can fire with any delay up to 1 time unit. The level-ruled Petri nets in this figure describe only part of the behavior

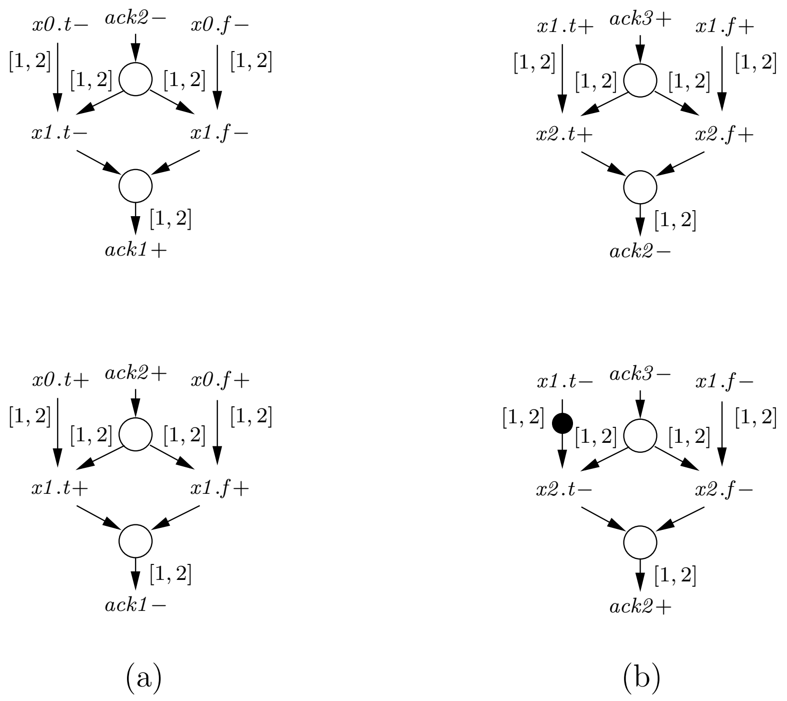


Fig. 2.15. The empty and full stage model for a STARI FIFO. (a) The level-ruled Petri net for the empty first stage. (b) The level-ruled Petri net for full second stage.

of the STARI FIFO. The complete behavior is defined in the parallel composition of the network of modules describing the system.

The parallel composition of the network of modules for two stages of the STARI FIFO is shown in shown in Fig. 2.16. This composition includes the nets in Fig. 2.14 and Fig. 2.15 that completely describe the environment and FIFO stages of the system. In the initial marking, (12,*clk*+) is the only enabled delay-transition pair to fire. The parallel composition restricts the behavior of the *clk* signal in the first and second stage models. The *clk* input is no longer unconstrained and must fire within the specified timing bounds. The firing of (12,*clk*+) results in a state where *clk*− can fire with a delay of 12, $x0.t$+ or $x0.f$+ can fire with a delay up to 1, and *ack3*− can fire with a delay between 1 and 2. The maximum allowed delay in this state is 1, so *clk*− can never fire before the other transitions. Assume that (0,$x0.t$+) fires first. This firing causes $x0.f$+ to no longer be marking satisfied as it consumes the token in the place they share in their preset.

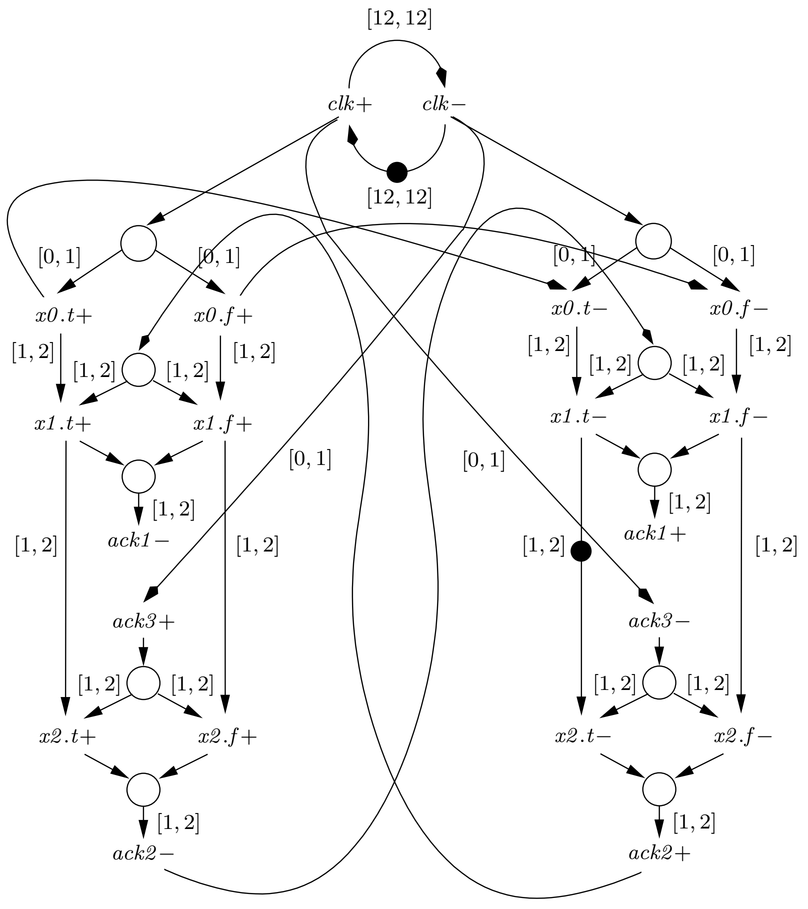


Fig. 2.16. The level-ruled Petri net for the composed STARI FIFO.

The STARI FIFO demonstrates the use of the level-ruled Petri net to describe systems with modules without using syntactic abstraction. It shows how the modules are composed to produce the parallel composition of the system. The final composed system can be analyzed, or it can be used in conjunction with one of its constituent modules to reduce the complexity of the analysis.

2.4 Related Work

The system model affects not only the ease of specification, but the cost of analysis too. The goal of analysis depends on its application. A model for verification may have different needs than one for synthesis. It is not uncommon to find several different ways to model a timed system; each one has its strengths and weaknesses according to the goal it is trying to achieve. The myriad of existing models makes selecting an appropriate model challenging.

Alur introduces timed automata as a signal model for timed systems in [18]. A *timed automaton* is a state based specification language where transitions between states are governed not only by Boolean functions defined over inputs, but clock valuations too. An example of a timed automaton model of the function $c = a \wedge b$ is shown in Fig. 2.17(a). Each node of the automaton is a state. This example has two states. Each node may be given an optional invariant label. This label can include not only Boolean logic values on the signal wires, but also allowed ranges of the clock variables. The example has an invariant label in each state. The first label forces the two functions to be equivalent. The second label allows the system to stay in the state as long as the clock C is under 45. Transitions are governed by the state of the inputs and the values of the clocks in the system. In this example, a transition from the left to right state is governed by X and F not being equivalent. There are two transitions from the right back to the left state. If the X and F become equivalent, or if they are not equivalent and the clock is between 35 and 45 time units. When a transition is taken, the system executes actions on clocks and signals in the system. In this example, either clock C is reset to 0, or the output X is inverted to match its function F. A system is the parallel composition of several timed automaton.

A timed automaton is a suitable model for verification as shown in [19, 20, 21, 22, 23]. Work by Bengtsson et al. in [24] and Minea in [25] develop partial order techniques to reduce the cost of verification in timed automata. Work by Bozga uses symbolic methods to help contain the verification cost [20, 26, 27, 28]. It is important to note that timed automata support both time and Boolean functions. They are very similar to level-ruled Petri nets. In a timed automaton, however, the designer must declare each of the clocks as part of the specification. The designer then has the ability to use the clocks where desired. This is not the case for the level-ruled Petri net. A designer does not have the ability to selectively choose where and how clocks are used. The timed automaton is thus more expressive than the level-ruled Petri net in this sense. The expressiveness, however, is not always required; thus, it can needlessly complicate the analysis problem. Forcing the designer to manage the clocks can be tedious too.

A Petri net is an alternative model to the timed automata. The Petri net is first introduced in [5] with surveys presented in [6, 7]. A common form of Petri net is the signal transition graph introduced by Chu in [29, 30] and independently by Rosenblum in [31]. The *signal transition graph* is a labeled safe Petri net. The labeling function maps transitions in the net to transitions on signals in the system. The M-net in [32] and I-net in [33] are variants of the Petri net that resemble the signal transition graph. Varshavsky introduced change diagrams in [34]. A *change diagram* is similar to a signal transition graph only it includes new types of arcs to implement starting behavior and disjunctive causality. Note that the disjunctive causality is not through Boolean functions, but through special types of edges in the net. Moon adds a notion of *don't care* and conditional behavior to the signal transition graph in [35]. The nets can ignore random switching on certain signals and resolve choice through Boolean functions. Ramchandani adds timing to the Petri net in [36] by breaking each transition into two transitions separated by a single place. When the first transition is enabled, it removes the tokens from its preset and places a token in its single postset place. The token then remains in that postset for some amount of time. The second piece of the transition then fires and puts places in its postset. The fixed amount of time can be random based on some distribution. Merlin adds delays to transitions in [37]. The work by Vanbekbergen in [38] is of particular interest. Vanbekbergen extends the signal transition graph to include minimum and maximum time intervals on places, four types of transitions, and Boolean guards on places with multiple transitions in their postset. The four types of transitions are: first, normal up and down transitions like those in the level-ruled Petri net ; second, a *don't care* transition that allows a signal to randomly toggle between high and low states; third, *level* transitions that indicate the Boolean state of the signal after the transition regardless of the state before the transition; and fourth, *toggle* transitions that complement the current state of the signal. Although this extension is expressive and similar in many ways to the level-ruled Petri net, Vanbekbergen does not present any type of analysis algorithms for it.

The event structure introduced by Winskel is a signal model that is similar to the Petri net and its many variants [39]. Burns adds fixed delays to the event structure and develops algorithms to compute the average-case performance of control implementations in [40, 41]. Myers added minimum and maximum bounds to the event rule structure in [42]. Belluomini extends the event rule structure to timed event/level structures in [10, 43]. An example of a timed event/level structure for the function $c = a \wedge b$ is shown in Fig. 2.17(b). This model strikes a keen resemblance to the level-ruled Petri net. Like the Petri net it includes a minimum and maximum delay bound coupled with a Boolean function on the edges. An important difference, however, is the absence of places. The timed event/level structure has no notion of a place; thus, all conflict must be resolve through a relation in the model. A simple Petri net conflict structure is shown in Fig. 2.18(a). The transition t_1 and t_2 share a common place in their preset; thus, only one of the two transitions can fire. The identical structure for the timed event/level structure is shown in Fig. 2.18(b). In this example, there are two rules (t_1, t_2) and (t_1, t_3). Each is marked as indicated by the tokens on the rules. The structure does not indicate that t_2 and t_3 conflict. This information is in the conflict relation and

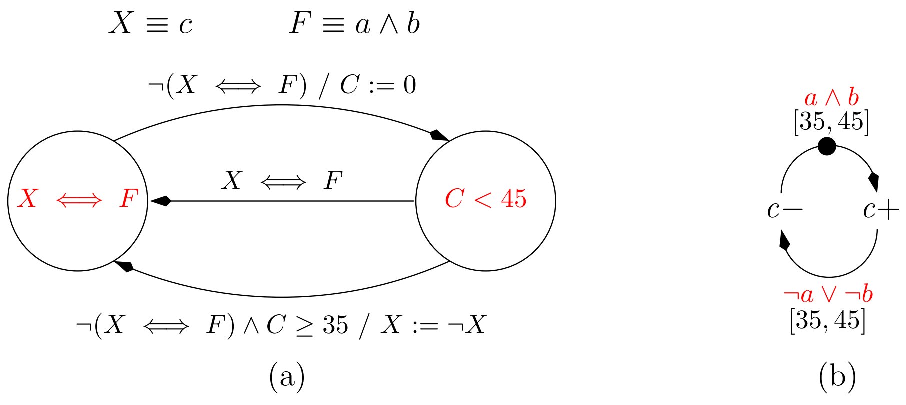


Fig. 2.17. Two representations of the function $c = a \wedge b$. (a) The timed automata representation. (b) The timed event/level representation.

denoted in the figure by the $t_2 \# t_3$ indication. Any arbitrary pair of events can be included in the conflict relation as shown in Fig. 2.18(c). Although transitions t_2 and t_4 are structurally independent, the timed event/level structure can make them conflicting through the relation. This generality makes the timed event/level structure so that it is not an equivalent model to the level-ruled Petri net. There are things that can be modeled in each system that cannot be modeled by the other system. The level-ruled Petri net, however, enjoys the formalism, semantics, and algorithmic support of the Petri net world.

There is another important difference between the level-ruled Petri net and the timed event/level structure: *disabling* and *nondisabling* semantic support for rules. Consider a rule that is currently marked and level satisfied. Its clock is increasing monotonically. Now suppose that a transition on a signal fires to move the system to a new Boolean state, and the rule is no longer level satisfied. It is, however, still marked. Now another signal transition fires to move the system back into a state where the rule is once again level satisfied. Disabling semantic support resets the clock for the rule back to zero at this point; and during the period where the rule is not level satisfied, it cannot contribute to the enabling condition of a transition. This is the defined semantics for the level-ruled Petri net . The clock on the rule behaves differently, however, in nondisabling semantics. When the rule is no longer level satisfied by the Boolean state, nondisabling semantic support still considers

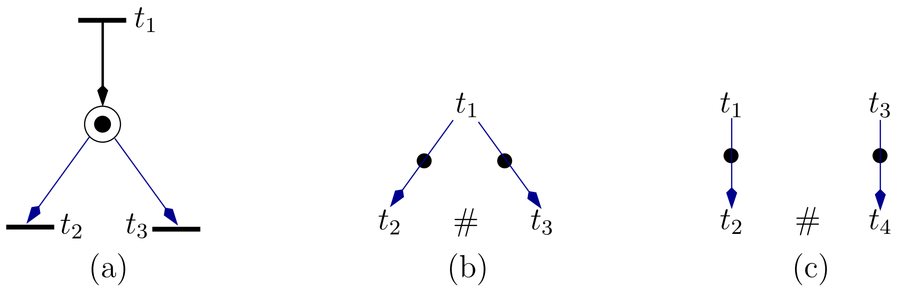


Fig. 2.18. Conflict structures in the Petri net and timed event/level models. (a) A Petri net conflict structure for two related transitions. (b) A timed event/level structure for two related conflicting transitions. (c) A timed event/level structure for two nonrelated conflicting transitions.

the rule active and contributing to the enabling condition of transitions. The rule is active until it is no longer marking satisfied. This implies that when the system moves to the new Boolean state where the rule is once again level satisfied, its clock is not reset because the rule is never deactivated. In brief, once a rule is marking and level satisfied, it becomes enabled and does not lose its enabling due to a change in Boolean state, only a change in marking. Nondisabling semantics are not currently supported in the level-ruled Petri net model.

2.5 Summary

This chapter presents the level-ruled Petri net as a model for timed circuit specification. The level-ruled Petri net supports minimum and maximum timing annotations on arcs between places and transitions, as well as Boolean functions. The Boolean functions are not restricted to only conflict places, but can be used to annotate any arc between a place and transition. The basis in the Petri net formalism allows it to model arbitrary concurrency. This chapter formally defines its structure and semantics. It presents definitions to compose a complete timed circuit system from a network of timed level-ruled Petri nets, including a definition of a valid composition for the final system.

This chapter presents unique examples of the level-ruled Petri net modeling various synchronous and asynchronous systems. The first is a pipeline from IBM that is optimized to reduce power and wire delay in traditional synchronous pipelines. The other example is another industrial design from IBM. It is a high performance pipeline that obtained unprecedented throughput and frequency. The final is a STARI asynchronous buffer to communicate between two synchronous domains. The examples strive to demonstrate the modeling power of the level-ruled Petri net and its flexibility. The level-ruled Petri net can be used to specify both standard and nonstandard gate design. It can be applied directly to the structural implementation of the system or a more behavioral description if desired. In addition, it has the ability to specify very broad environments to drive the circuit models. These environments can be deterministic or random. The random environments produce all possible input behaviors of the circuit. An analysis algorithm can thus determine correctness of a design under all possible input conditions using the randomness in the environment.

This chapter presents a brief overview of related work. Although it is not comprehensive, it serves to give perspective to the level-ruled Petri net relative to other models. Although the level-ruled Petri net can be used as the primary specification language, it is supported by higher level languages too. A system can thus be described at a higher level and then be compiled to the level-ruled Petri net in a manner similar to that in [9].

CHAPTER 3

CORRECTNESS

Analysis has no meaning without a definition of correctness. This is true in timed circuit applications. An analysis algorithm can be applied to a timed circuit model, and it can report information from the analysis. The question is, however, what do the results imply? The implication is readily clarified through a formal definition of correctness in the model. The correctness definition enables an analysis algorithm to state the condition of the model. If correct, then the designer exactly understands what the circuit does in the model to make it correct. If incorrect, then the designer exactly understands what the circuit does in the model to make it incorrect.

The formal correctness definition is required to legitimate the timed circuit model. Remember that an analysis algorithm can only operate on behaviors present in the timed circuit model—the level-ruled Petri net in this work. Without the correctness definition, it is not clear that the level-ruled Petri net is the appropriate model for the analysis. If a designer is looking to explore a property that is outside the scope of the level-ruled Petri net, then the analysis results are of no import to the designer. More importantly, however, is that the designer may believe a circuit to have certain properties that are never considered. The precise definition of correctness in the level-ruled Petri net removes any confusion in the meaning of the results; thus, the designer can select the appropriate model and analysis algorithm to validate properties of interest to the timed circuit application.

The cost of analysis is an important consideration in addition to the modeled properties. A rich set of properties and behaviors in the model adds to the cost of analysis. A model must carefully balance the present behaviors and the cost of analysis. This cost is reflected in the correctness properties in the model.

The correctness definition is given in terms of a component operating in a system environment. A complete system is often composed of several modules, each modeled by an appropriate level-ruled Petri net. Correctness considers the behaviors of a component in the parallel composition of the system. The level-ruled Petri net model of a component in the parallel composition of the system is correct if it is *safe*, *consistent state assigned*, *output semimodular*, and *constraint satisfied.* The safe property does not allow transitions to add existing places in the marking, and it simplifies timing semantics. The consistent state assignment property forces a single transition to fire between states toggling the Boolean state if the transition is on a signal, and it simplifies the semantics of the model. The output semimodular property prevents signals from disabling outputs or outputs disabling any transitions visible to the component. The constraint satisfied property, finally, checks user specified timing and ordering requirements in the model.

The safe, output semimodular, and consistent state assignment properties are directly supported in the level-ruled Petri net. These are common correctness properties in Petri net based models. The constraint satisfied property is not directly supported, and it is a less common approach to correctness. Section 3.1 presents an extension to the level-ruled Petri net to support the constraint satisfied property. All of the correctness properties are then formalized in the level-ruled Petri net semantics in Section 3.2.

The goal of this chapter is to make a definitive statement on correctness in the timed circuit model. Section 3.3 formalizes this statement and presents algorithmic requirements necessary to show correctness in level-ruled Petri net analysis. A survey of related work is presented in Section 3.4. This chapter is concluded with a brief summary of salient points in Section 3.5.

3.1 The Constraint Rule

The constraint rule is a mechanism to refine the level-ruled Petri net structure and semantics. The refinement enables the user to specify bounded timing response and required transition orders in the level-ruled Petri net. These specifications can be made between any two transitions in the system regardless of their causal relation. Consider again the STARI FIFO from Section 2.3.3 in Chapter 2 and for convenience, shown here again in Fig. 3.1. The STARI FIFO enables communication between two circuits that are operating at the same clock frequency but are out-of-phase due to clock skew [16, 17]. There are two properties that need to hold in a correct STARI FIFO: first, each data value output by the transmitter must be inserted into the FIFO before the transmitter sends another data value; and second, a new data value must be output by the FIFO before each acknowledgment from the receiver [44]. These two properties are modeled using constraint rules.

A constraint rule defines a requirement on the order and time separation between transitions. Consider the level-ruled Petri net model with constraint rules for the first stage of the STARI FIFO in Fig. 3.2(a). The constraint rules are shown as dashed edge connections between places and transitions. A constraint rule is different from an ordinary rule because it does not affect the behavior of a transition. The enabling condition of a transition is determined completely by ordinary rules. Once a transition is selected to fire, however, constraint rules are checked to see if the firing violates user defined timing and ordering requirements. The constraint rules in Fig. 3.2(a) check the first property of the STARI FIFO. The *x0.t*+ or *x0.f*+ inputs from the transmitter TX must always come after the *ack1*+ transition from the first stage of the FIFO. Furthermore, these inputs must not arrive earlier than 3 time units after the *ack1*+ transition. This checks the first property of the

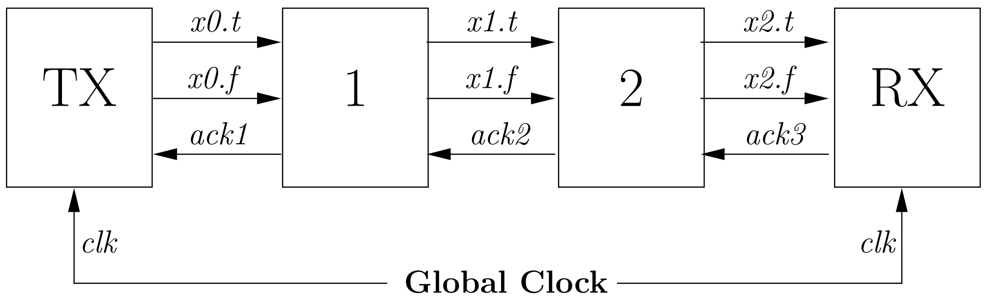


Fig. 3.1. The block diagram of a dual-rail STARI FIFO with two stages.

STARI FIFO because the *ack1*+ transition indicates that the first stage is ready for another data item; thereby, it implies that the previous data item has been inserted into the FIFO. The second property is checked on the second stage of the FIFO in a similar fashion as shown with the constraint rules in its model in Fig. 3.2(b). The *x2.t*+ or *x2.f*+ transition must precede the *ack3*− input transition from the receiver RX within 9 to 13 time units as expressed by the constraint rule. This satisfies the second property because the *ack3*− transition indicates that the receiver has captured the data; thus, *x2.t*+ or *x2.f*+ must fire before *ack3*− to make the captured data valid. In order to satisfy this property, however, the STARI FIFO must be initialized to be half-full [17]. This is why the second stage of the FIFO initially contains a single data item. The receiver captures this data item on the first clock cycle.

The model of the STARI FIFO with two stages does not use syntactic abstraction. A constraint rule in an example with syntactic abstraction can have a unique

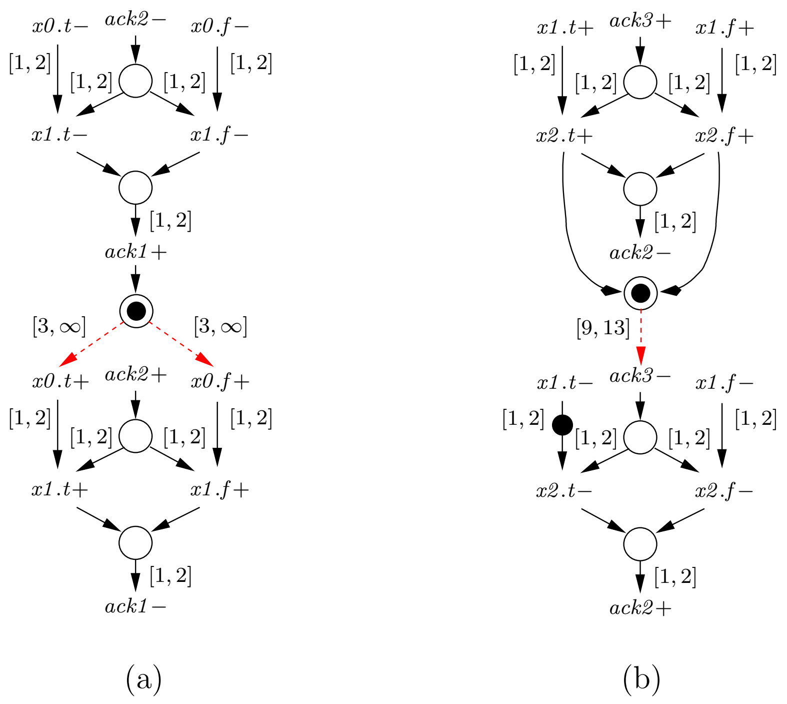


Fig. 3.2. The empty and full stage model for a STARI FIFO with constraints. (a) The level-ruled Petri net with constraint rules for the empty first stage. (b) The level-ruled Petri net with constraint rules for full second stage.

form. Consider the delayed-reset domino gate for the function $(a \vee b) \wedge c$ from Section 2.3.2 as shown in Fig. 3.3(a). An important property in this circuit is that the gates do not begin to precharge before their inputs have gone low. Consider the second stage of the gate in Fig. 3.3(a) that computes $f_2 = f_1 \wedge c$. The inputs f_1 and c must be low before the clk_2- transition to prevent a short circuit in the gate. This property can be checked using constraint rules. The model for the clk_2 input is shown in Fig. 3.3(b). It now includes a new constraint rule. This is the dashed rule. The rule is initially marked, and it is newly marked each time clk_2- fires. The constraint rule checks a minimum separation between the time when f_1- or $c-$ fire and the firing of clk_2- ensuring that the inputs to stage two are low a suitable time before the clk_2- transition. Constraint rules in models using syntactic abstraction often take this self-loop form.

Recall that a level-ruled Petri net is the tuple $M = (N, E)$, where $N = (T, P, F, \mu_o)$ is an ordinary Petri net, and $E = (W, L, \nu_o, \mathsf{Eft}, \mathsf{Lft}, \mathsf{Lsat})$ is a level-ruled extension for the net N (Definition 2.10). The set of rules in M is given as $R = F \cap (P \times T)$. The set of constraint rules is defined as $C \subseteq R$; thus, it is a subset of the rules in R. Definition 2.8 defines a function that takes a transition and returns the set of rules connected to that transition. This definition is refined to reduce the number of structure and semantic definitions that must be revisited

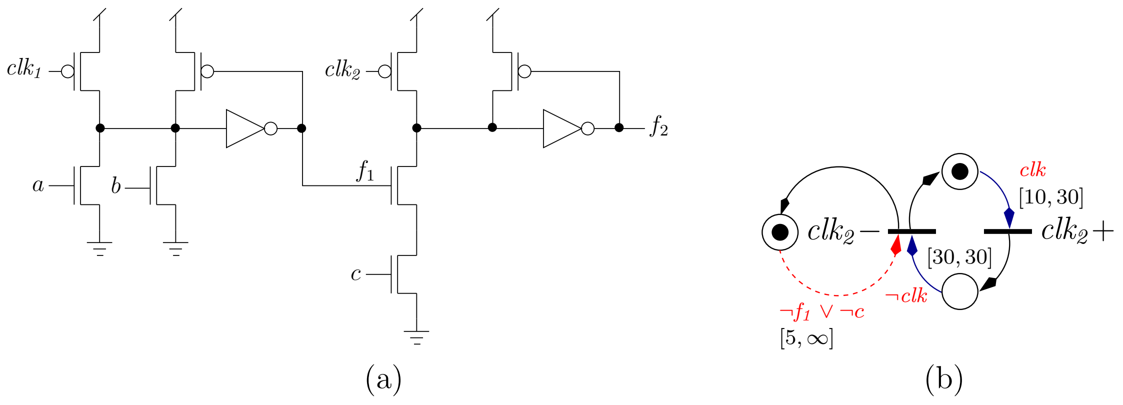


Fig. 3.3. A two-stage delayed-reset domino gate and its *clk2* model. (a) A two-stage delayed-reset domino gate that computes $(a \vee b) \wedge c$. (b) The level-ruled Petri net model of the *clk2* input with a setup constraint.

to support constraint rules.

Definition 3.1 (Ordinary Rule Set). *For any transition $t \in T$, the rule set of t is given as $R(t) = \{(p,t) \in R \mid p \in \bullet t \wedge (p,t) \in (R - C)\}$.*

This definition is different from Definition 2.8 in that it only includes rules that are not found in the constraint set C. A new function to return constraint rules in a set of places follows next.

Definition 3.2 (Constraint Rule Set). *For any transition $t \in T$, the constraint rule set of t is given as $C(t) = \{(p,t) \in C \mid p \in \bullet t \wedge (p,t) \in C\}$.*

This function returns all rules defined in the flow relation as connected to a transition that are constraint rules. With this refinement and new definition, the notion of constraint rules can be approached in the structure and semantic definition of the level-ruled Petri net.

The goal of this section is to present the structure and semantic changes to the level-ruled Petri net to support the constraint rules. This is done by refining existing definitions; Section 3.1.1 parallels Section 2.2.1, and Section 3.1.2 parallels Section 2.2.2. In each section, only affected definitions are refined. All other definitions remain unchanged.

3.1.1 Structure

The structure of the level-ruled Petri net in Definition 2.9 is augmented to include the set of constraint rules C.

Definition 3.3 (Structure). *A level-ruled Petri net with constraints is the three-tuple $M = (N, E, C)$ consisting of a Petri net N, its level-ruled extension E, and a set of constraint rules C.*

The Petri net N in Definition 2.1 and its level-ruled extension E in Definition 2.10 remain unchanged, as well as the output set O in Definition 2.11.

The parallel composition of a network of level-ruled Petri nets must now include the set of constraint rules in the composition.

Definition 3.4 (Parallel Composition). *The parallel composition of a network of level-ruled Petri nets $M_1 \parallel M_2 \parallel \cdots \parallel M_n$ creates the single net $M = (N, E, C)$ where for $1 \leq i \leq n$ and $M_i = (N_i, E_i, C_i)$, N is the union over the constituent members of each N_i; C is the union over each constraint set C_i; and E is the union over the constituent members of each E_i excepting* Lsat, *which is the new function such that for all indices $j \in \mathbb{N}$, rules $r_j \in R_j$, and Boolean states $\nu \in 2^W$, $1 \leq j \leq n$ and*

$$\mathsf{Lsat}_j(r_j)(\nu') = \mathsf{true} \iff \mathsf{Lsat}(r_j)(\nu) = \mathsf{true},$$

where $\nu' = \nu \cap W_j$ is the Boolean state ν with all of the signals not in the component signal set W_j removed and $\mathbb{N}$ is the set of natural numbers.

For the parallel composition to be valid, however, the network must be consistent in the definition of the constraint rules in their intersection.

Definition 3.5 (Consistent Composition). *A given pair of level-ruled Petri nets (M, M') is consistent if they satisfy Definition 2.13 and for all rules $r \in (R \cap R')$, $r \in C \iff r \in C'$.*

Two nets are consistent if, from Definition 2.13, they agree on the marking of shared places, the Boolean state of shared outputs, and identically define shared rules. Added to this definition is that the two nets must be consistent in their definition of constraint rules. Given these refinements, Definition 2.14 correctly formalizes a valid composition in the presence of constraint rules.

3.1.2 Semantics

A constraint rule is different than an ordinary rule because it does not affect the behavior of a transition. In firing a transition, only rules in $(R - C)$ affect the transition firing. Once a transition is selected to fire, however, the rules in C can be checked to see that the transition does not violate user defined timing and ordering requirements from the specification. This is formalized in Section 3.2.4. This section, however, performs the necessary refinement in the level-ruled Petri net semantics to support constraint rules as described.

The semantic definition of the level-ruled Petri net requires one refinement to support constraint rules because constraints rules are passive participants in the model. Their role, as mentioned earlier, is to observe, not to affect transitions. As such, they are by default ignored in all existing definitions with the refinement given in Definition 3.1. This works correctly in all instances but Definition 2.20.

Definition 2.20 formalizes the notion of a set of rules being time satisfied by a clock assignment function. The definition allows clocks on rules to exceed their upper bounds as long as a single rule exists in the rule set whose clock is below its upper bound in the clock assignment function. Although these semantics are appropriate for rules that affect the behavior of transitions, they are not correct for constraint rules that operate as observers. The timing on constraint rules in the model indicates bounded response by the model. The value of the clocks for all constraint rules on a transition must be below their latest firing time when the transition fires for this to be true. Definition 2.20 is refined to reflect this notion.

Definition 3.6 (Time Satisfied with Constraints). *The rule set R' is time satisfied by the clock assignment function $\mathcal{C}$ if for all rules $r \in R'$, $\mathcal{C}(r) \geq \mathsf{Eft}(r)$; and either of the two following conditions hold:*

1. *$R' \neq \emptyset$, $R' \subseteq (R - C)$, and there exists a rule $r \in R'$ such that $\mathcal{C}(r) \leq \mathsf{Lft}(r')$; or*

2. *$R' \subseteq C$ and for all rules $r \in R'$, $\mathcal{C}(r) \leq \mathsf{Lft}(r)$*

$\mathcal{C} \vdash R'$ indicates that the property holds and $\mathcal{C} \nvdash R'$ the opposite.

Definition 3.6 now implements semantic support for constraint rules. A rule set that contains rules only from $R - C$ is time satisfied if its clocks are above their earliest firing time in the clock assignment $\mathcal{C}$ and there exists a single rule in the set whose clock assignment in $\mathcal{C}$ is below its latest firing time. This follows Definition 2.20. A rule set that contains only rules from C, however, is time satisfied if its clocks satisfy both their earliest and latest firing times. Note that any mix of constraint and ordinary rules in the rule set makes it not time satisfied regardless of the clock valuations in $\mathcal{C}$.

3.2 Correctness Properties

Correctness properties define the set of accepted behaviors in a level-ruled Petri net of a component in a network. The term *accepted* as used here refers specifically to behaviors that are deemed correct in the actual timed circuit application and its environment. The goal of the level-ruled Petri net model of the timed circuit is to explore the behaviors of the circuit and confirm the absence of inappropriate behavior in the defined environment; thus, the correctness properties do not force the model to be correct. They simply serve to identify situations where it is incorrect, meaning, it violates a correctness property.

This section formally defines the safe, consistent state assigned, output semi-modular, and constraint satisfied correctness properties in the level-ruled Petri net model. Each property is presented in its own section along with a simple example for illustrative purposes.

Notation must be presented before the correctness properties are defined. First, the $[\ ]$ operator denotes the length of a vector in this presentation. Consider the vector of transitions $\mathbf{t} = (t_1, t_2, \ldots, t_n)$, $[\mathbf{t}]$ is equal to n, where n is a natural number of any size. The length of the empty vector ϵ is zero. Second, recall from Section 2.2.2 that a state of a level-ruled Petri net is the state tuple $s = (\mu, \nu, \mathcal{C})$. The symbol s or $(\mu, \nu, \mathcal{C})$ can be interchanged; both represent the same thing. Given a state s, the symbols μ, ν, and $\mathcal{C}$ refer to the members of the state tuple s. If the given state appears primed (s') or subscripted (s_i) then members of the state tuple also appear appropriately primed (μ',ν',$\mathcal{C}'$) or subscripted (μ_i,ν_i,$\mathcal{C}_i$). Third, the left-hand argument of the $\vdash$ and $\nvdash$ operator can be a tuple. This is seen in Definition 2.22 where $\mu \vdash R'$, $\nu \vdash R'$, and $\mathcal{C} \vdash R'$ is indicated by $(\mu, \nu, \mathcal{C}) \vdash R'$. $(\mu, \nu, \mathcal{C}) \vdash R'$ holds if $\vdash$ holds when applied to each member of the tuple using the same right-hand argument—a conjunctive relation. The $\nvdash$ operation extends to a left-hand tuple argument, only it holds if any members of the tuple when paired with the right-hand argument do not hold—a disjunctive relation. For example, $(\mu, \nu, \mathcal{C}) \nvDash R'$, if either $\mu \nvDash R'$, $\nu \nvDash R'$, or $\mathcal{C} \nvDash R'$. Fourth and final, all the correctness properties are presented in terms of a component in a larger system of level-ruled Petri nets. As such, it is assumed that the global net, which is the valid parallel composition of all the nets in the system, is visible. The global net is given as $M = (N, E, C)$ from Definition 3.3 where $N = (T, P, F, \mu_o)$ is the Petri net from Definition 2.1, $E = (W, L, \nu_o, \mathsf{Eft}, \mathsf{Lft}, \mathsf{Lsat})$ is the net's level-ruled extension from Definition 2.10, and C is the set of constraint rules. Anytime these symbols appear without a subscript or prime, they refer to the corresponding members of M. Similarly, all states s, with or without subscripts or primes, are defined over the system level model M unless otherwise noted in the text.

3.2.1 Safety

The level-ruled Petri net model of a component in a system of nets is safe if it never tries to add more than one instance of any of its places to the marking. A violation of the safe property often indicates a problem in the specification. The safe property also simplifies analysis because the marking no longer needs to track the number of times a place appears. Finally, it simplifies the semantics of the level-ruled Petri net because it is not necessary to define which instance of a place needs to be used first.

The safe property increases the probability that a circuit to implement the model exists. Imagine that the actual circuit for a component has yet to be created. The designer does, however, have a specification for the component. A level-ruled Petri net model of the component is thus created according to the specification. An analysis algorithm can then try to synthesize a circuit from the reachable state space of the component model in the system. The likelihood of a circuit existing to implement the model is increased if the model is safe in the environment.

An example of the safe property is seen in the STARI FIFO. Fig. 3.4 is the level-ruled Petri net model of the STARI FIFO in Fig. 3.1. This is the same model presented in Section 2.3.3. This figure does not include the constraint rules introduced in Fig. 3.2(a) and Fig. 3.2(b) to simplify the presentation of the safe

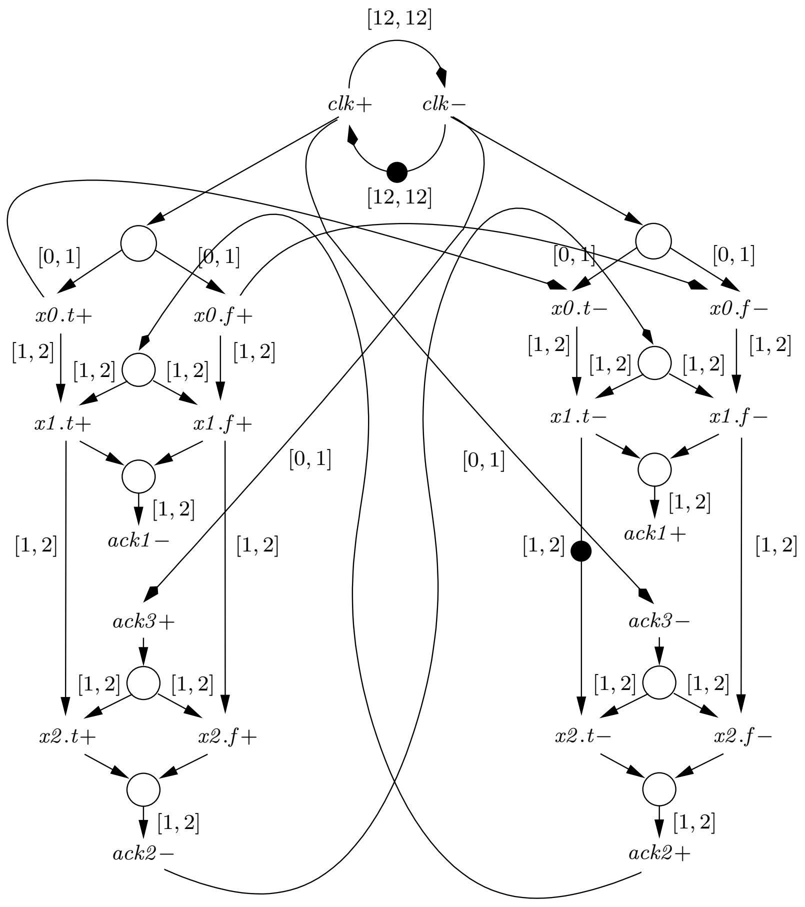


Fig. 3.4. The level-ruled Petri net for the composed STARI FIFO.

property. The *clk* signal cycles every 24 time units in this model. The TX module sets and then resets either *x0.t* or *x0.f* every 24 time units too. Consider the scenario where the TX module inserts into the FIFO two consecutive *x0.t* data items. At the first *x0.t+* transition, the first stage of the FIFO needs to fire its *x1.t+* transition in response to its newly arrived input. This transition can fire once it is marking and time satisfied, which happens after its other input transition *ack2+* fires followed by at least 1 time unit of delay. The *ack2+* transition cannot fire until the RX module fires *ack3−* followed by the second stage firing *x2.t−*. If the delay to fire *ack3−* followed by *x2.t−* and then *ack2+* plus 1 time unit for the *x1.t+* transition to become time satisfied is greater than the clock period, then the second *x0.t+* transition can fire before the first stage responds to its prior firing. The first stage of the STARI FIFO is not safe in the environment if this situation can occur because nothing responds to the marked place of *x0.t+* before a transition tries to mark it again. A safety violation implies that the circuit receives an input pulse on a signal, but before it responds to that input pulse, it receives another input pulse on the same signal. The circuit is not able to remember that it has received two input pulses because it never responded to the first input pulse; thus, it only produces a single output pulse.

A safety failure in a model using syntactic abstraction is identical to the failure shown in the STARI FIFO. The model with the abstraction does not require a unique or different form to violate the safe property. If the component models of a system all have a structure similar to the one shown in Fig. 3.3(b), however, then they can never fail the safe property. This type of structure can never try to add a multiple instance of a place to the marking because all of the models have a loop structure. This is a very common form in specifications using syntactic abstraction. To have a failure of the safe property, structures similar to those in the STARI FIFO are required.

The safe property is a dynamic characteristic of the level-ruled Petri net. Although it is a structural property of the ordinary Petri net, the addition of time changes this in a level-ruled Petri net. If the STARI FIFO model ignores time, then the first stage model is clearly not safe because the delay just to fire *ack2+* can be of an arbitrary size; thus, there surely exists a firing sequence where either of the *x1.t* or *x1.f* transitions do not respond to their marked place before the place is marked again. The time requirements in the model, however, can make it safe.

The safe property is a function of the timed circuit model in the reachable states of the system model.

Definition 3.7 (Safe Transition). *A transition $t \in T$ is safe in the marking μ for a given set of places $P' \subseteq P$ if $((\mu \cap P') - \bullet t) \cap (t\bullet \cap P') = \emptyset$*

Note that Definition 3.7 is restricted to only consider places in $P' \subseteq P$ as the safe property only applies to a component in a larger system. Recall from Definition 2.31

that a firing sequence is a pair consisting of a delay vector and a transition vector $(\mathbf{d}, \mathbf{t})$ of equal length; and a state s is the tuple $(\mu, \nu, \mathcal{C})$

Definition 3.8 (Safe Firing Sequence). *A firing sequence* $(\mathbf{d}, \mathbf{t})$ *is safe for a given set of places* $P' \subseteq P$ *if in its corresponding state vector* $\mathbf{s}$*,* t_i *is a safe transition in* μ_{i-1} *given* P' *for all indices* i *such that* $1 \leq i \leq [\mathbf{t}]$*.*

A firing sequence is safe if all of its transitions are safe in its corresponding state vector when restricted to consider only places in P'. Recall that $\mathcal{P}(s_o)$ is the set of all possible firing sequences starting from the initial state $s_o = (\mu_o, \nu_o, \mathcal{C}_o)$.

Definition 3.9 (Safe Component). *A component* M_i *in a network of level-ruled Petri nets* $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ *is safe if for all firing sequences* $(\mathbf{d}, \mathbf{t}) \in \mathcal{P}(s_o)$ *of* M*,* $(\mathbf{d}, \mathbf{t})$ *is safe when given the place set* P_i *from the component* M_i*.*

Definition 3.9 is verified by an exploration of the reachable states. Definition 2.7 is the safe property for Petri nets. It is verified through structural analysis of the net. The addition of time prevents the use of structural analysis to verify the safe property in level-ruled Petri nets. Time may make a net safe as it precludes some transition orders. Note that Definition 3.9 includes constraint rules in the safe property as the property is defined using the preset and postset of the transitions. Places are included in the preset and postset sets regardless of their rule status.

Consider again the STARI FIFO network in Fig. 3.4. If the delay from *ack3*$-$ to *ack2*$+$ is longer than the clock period through any path, then there exists a firing sequence where either *x0.t*$+$ or *x0.f*$+$ fires to insert into the marking a place that already exists from the previous firing. If this sequence exists, then the first stage model is not safe.

3.2.2 Consistent State Assignment

A component of a system has a consistent state assignment, if all of its signals strictly toggle state, and any adjacent states in a firing sequence do not differ in more than one signal value. Consider the first stage model of the STARI FIFO in Fig. 3.2(a). The *x0.t*$-$ transition is an unconstrained input in this model. Any firing sequence that includes two *x0.t*− transitions without an intervening *x0.t*+ transition is not consistent state assigned. This property simplifies the semantics of the level-ruled Petri net because it is not necessary to define model behavior for firing transitions on signals that are already at the correct final state. The simplified semantics in the level-ruled Petri net reduce the cost of analysis.

A violation of the consistent state assigned property is often the result of a mistake in the model. The is most often true when using syntactic abstraction. Consider again the level-ruled Petri net model of the clk_2 signal in Fig. 3.3(b) for the delayed-reset domino gate in Fig. 3.3(a). If the initial state, ν_o, for the system shows the clk_2 signal to be high, then the model is not consistent state assigned. The clk_2+ transition can fire once it is level and time satisfied in the shown marking causing a consistent state violation. This violation only exists because of the incorrect initial Boolean state.

The consistent state assigned property is first defined in terms of a transition firing from a state.

Definition 3.10 (Consistent Transition). *A transition $t \in T$ is consistent in the marking and Boolean state (μ, ν) if $(L(t) = w+ \implies w \notin \nu) \wedge (L(t) = w- \implies w \in \nu)$.*

A marking and level satisfied transition is consistent if it does not affect the state of any signal, or if it strictly toggles the state of a signal.

The consistent state assigned property applies to a component in a larger system, like the safe property. This is not reflected in Definition 3.10. The scope of the consistent property is restricted when looking at a firing sequence.

Definition 3.11 (Consistent Firing Sequence). *A firing sequence $(\mathbf{d}, \mathbf{t})$ is consistent for a given set of transitions $T' \subseteq T$ if in its corresponding state vector $\mathbf{s}$ the following implication holds for $1 \leq i \leq [\mathbf{t}]$:*

$$t_i \in T' \implies t_i \textit{ is a consistent transition in } (\mu_i, \nu_i).$$

The set $T' \subseteq T$ restricts the scope of the consistent definition to members of

T'. A firing sequence is consistent if every transition from T' is consistent in the corresponding state vector. A required property in a consistent firing sequence is that adjacent state codes differ in at most one place. This is checked in the definition because the implication forces $s_{i-1}\,[(d_i, t_i)\rangle\, s_i$ to hold. If it does hold and the transition t_i is consistent from ν_{i-1}, then by Definition 2.23, the Boolean state code in the two states (s_{i-1}, s_i) can differ only in at most one signal value if the transition is on a signal in the system. If the transition is not defined on a signal, then the state codes are identical.

Definition 3.12 (Consistent State Assignment). *A component M_i in a network of level-ruled Petri nets $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ has consistent state assignment if for all firing sequences $(\mathbf{d}, \mathbf{t}) \in \mathcal{P}(s_o)$ of M, $(\mathbf{d}, \mathbf{t})$ is consistent when given the set of transitions T_i from the component M_i.*

A level-ruled Petri net is consistent state assigned if every transition in the module is consistent in every appearance in all firing sequences.

3.2.3 Output Semimodular

Output semimodular is a property of a circuit in an environment. A level-ruled Petri net model of a timed circuit component is output semimodular if in its defined environment, an output, once enabled, remains enabled until it fires, and firing an enabled output does not disable any transitions visible to the component. This ensures that a component's output in the defined environment does not generate runt pulses and does not require any arbiters in its implementation.

Consider the delayed-reset domino gate for the function $(a \vee b) \wedge c$ from Section 2.3.2 as shown in Fig. 3.3(a). If the input c and the function $f_1 = a \vee b$ are high, then the output f_2 can transition high after an appropriate delay. Consider this same scenario in the level-ruled Petri net model of the f_2 function in Fig. 3.5. The transition f_2+ is marking and level satisfied. The transition must fire before the expression on its rule, $f_1 \wedge c$, becomes false. If it does not fire before the expression becomes false, then its output is undefined because the signal f_2 is not in a discrete state; thus, the output may show a glitch. If the models for f_1- or $c-$ allow these

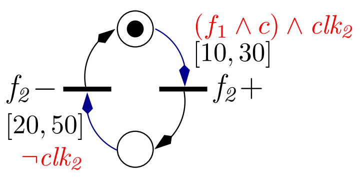


Fig. 3.5. The second stage level-ruled Petri net model to compute f_2.

transitions to fire too early, or if f_2+ is too slow, then the component computing f_2 is not output semimodular in its environment.

Outputs can be disabled through a change in marking or Boolean state. In the previous example, the f_2 output is disabled through a change in the Boolean state. Consider the level-ruled Petri net fragment in Fig. 3.6(a). This fragment is derived from the parallel composition of a network of level-ruled Petri nets. The interface for a component of this network is shown in Fig. 3.6(b). It has inputs a and b and a single output c. Suppose that the state of Fig. 3.6(a) is such that transitions $b+$ and t are marking, level, and time satisfied and that transition t fires to move the state of the net into a new marking. The transition $c+$ is now marking and level-satisfied. If $c+$ can become time satisfied and fire before $b+$, then the component in Fig. 3.6(b) is not output semimodular in this environment definition. The output $c+$ disabled the input $b+$, which is a transition visible to the component.

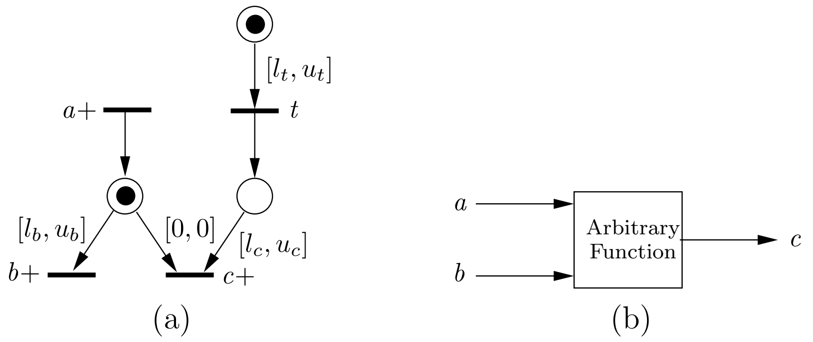


Fig. 3.6. A net fragment of a composition with a member component interface. (a) The level-ruled Petri net fragment from the parallel composition of a network of models. (b) The interface of a member component with inputs a and b and an output c

A component in a network of level-ruled Petri nets is shown to be output semimodular in the parallel composition of the network by analyzing the reachable states allowed by the system. Recall that $(\mu, \nu) \vdash R(t)$ denotes that transition t is marking and level satisfied by μ and ν from Section 2.2.2 and Definition 2.21, respectively. The semimodular property is first formalized in terms of a delay-transition pair leading the system from one state to another.

Definition 3.13 (Semimodular Transition). *A transition t is semimodular in the state pair (s_i, s_j) for a given set of visible transitions $T_V \subseteq T$ and a set of transitions on outputs $T_O \subseteq T_V$ if the following implication holds: $(\mu_i, \nu_i) \vdash R(t)$ and $(\mu_i, \nu_i)\,[t\rangle\,(\mu_j, \nu_j) \implies$*

1. *$t \in T_O \wedge ((\mathsf{mls}(\mu_i, \nu_i) - \{t\}) \cap T_V \subseteq \mathsf{mls}(\mu_j, \nu_j) \cap T_V)$; or*

2. *$t \notin T_O \wedge ((\mathsf{mls}(\mu_i, \nu_i) - \{t\}) \cap T_O \subseteq \mathsf{mls}(\mu_j, \nu_j) \cap T_O)$;*

where for a given $s = (\mu, \nu, \mathcal{C})$, $\mathsf{mls}(\mu, \nu) = \{t \in T \mid (\mu, \nu) \vdash R(t)\}$.

Definition 3.13 considers two special sets T_V and T_O to determine if a transition is semimodular. The first set T_V is the set of transitions that are visible to a component. The second set T_O is the set of transitions associated with output signals for the component. The second set is a subset of the first set. The definition considers a pair of states and a single transition. Two conditions must hold for a marking and level satisfied transition to be semimodular. The first condition is for transitions in the output set. The function $\mathsf{mls}(s)$ returns the set of transitions that are marking and level satisfied in the state s. If the transition is for an output signal, then the marking and level satisfied transitions in the first state s_i must be included in those in the second state s_j excepting the fired transition; thus, the firing of an output transition cannot disable any transitions that are visible to the component. The second condition is for transitions that are not on output signals. In this case, the set of output transitions that are marking and level satisfied in the first state s_i must be included in those of the second state s_j excepting the fired transition; thus, output transitions can never be disabled after they become enabled until they fire. If a transition meets the right hand side of the implication, then it is semimodular for the pair of states. Note that a pure environment module generating random inputs, such as the TX module from the STARI FIFO, is not output semimodular. Outputs are always disabled due to the random choice; thus, the output semimodular property is not required in pure environment modules.

Consider again the f_2 function for the delayed-reset domino gate in Fig. 3.5 and the following state pair. The first state is such that f_2+ and $c-$ are enabled with f_1 currently low in the Boolean state. The second state is the one created from firing $c-$ with some enabled delay. In this case, the transition leads the system from one state to the other. The right hand side of the implication, however, is not satisfied. The first condition is false because $c-$ is not an output. The second condition is false because the firing of $c-$ disables f_2+, which is an output. Consider now the level-ruled Petri net fragment in Fig. 3.6(a) and the following state pair. The first state is one such that $b+$ and $c+$ are enabled. The second state is the one after $c+$ fires on some enabled delay. In this example, the firing of the transition from the first state leads to the second state in the pair. The first condition, however, fails because the firing of the output transition $b+$ disables the visible transition $c+$; the second condition is false because $c+$ is an output.

Not all output semimodular violations can be detected in the dynamic behavior of the system as the above examples suggest. Consider the level-ruled Petri net fragment in Fig. 3.7(a). The transitions t_1 and t_2 can both fire in the current state. The firing of either transition disables the other. Suppose that in the component, transition t_3 is on a visible input signal and transition t_4 is on a visible output signal. Transitions t_1 and t_2 belong to other components and are not visible. The graph in Fig. 3.7(b) represents two possible vectors that the system can move through depending on the firing sequence. The edges represent transitions and the nodes represent states; s_0 of the graph relates to the current state of the net. The firing of transition t_1 leads to the state s_1 where t_3 fires. The firing of t_2 leads to the state s_2 where t_4 fires. Remember that transitions t_1 and t_2 are invisible to the component. Now consider a circuit's perspective for the component in this system.

It can only see the Boolean state of the signals on its interface. The associated markings and clock functions have no meaning. Fig. 3.7(c) shows the Boolean states that are perceived by the circuit. States s_0, s_1, and s_2 all look like the same state, and either t_3 or t_4 can fire in the state. The firing of output t_4 disables the input t_3; thus, the component is not output semimodular. This violation cannot be dynamically detected because it is not trace based. There does not exist a state in the reachable state space where both t_3 and t_4 are marking and level satisfied. This failure can only be detected after enumerating the timed states of the system. The interested reader in directed to Appendix A for complete details.

The set of transitions on signals visible to a target component is defined to support the output semimodular property.

Definition 3.14 (Transitions on Signals). *The set of transitions defined on signals in the set* $W' \subseteq W$ *is given as:* $T(W') = \bigcup_{w \in W'} T(w)$*, where* $T(w) = \{t \in T \mid L(t) = w+ \vee L(t) = w-\}$

The set $T(W')$ returns all transitions defined on signals in W' according to the labeling function. The visible transitions in module i in a network of modules are now given by $T(W_i)$, and its visible output transitions are given by $T(O_i)$. The

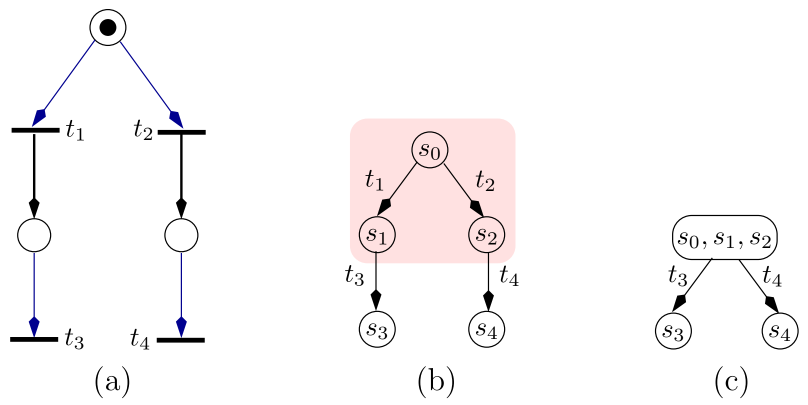


Fig. 3.7. A net fragment with its full and reduced state space. (a) A level-ruled Petri net fragment that enables two invisible transitions t_1 and t_2 on a choice place eventually followed by the visible transitions t_3 and t_4. (b) The state graph for the net fragment. (c) A reduced state graph for the visible transitions t_3 and t_4 showing the output semimodular violation.

output semimodular property is now formalized.

Definition 3.15 (Output Semimodular). *A component M_i in a network of level-ruled Petri nets $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is output semimodular if all transitions $t \in T$: for all state pairs $(s, s') \in [s_o\rangle$, there exists a delay $d \in \mathbb{R}^+ \cup \{\infty\}$ such that $s_i \vdash (d, R(t))$ and $s_i\, [(d,t)\rangle\, s_j \implies t$ is a semimodular transition on (s, s') given T_V and T_O, where $T_V = T(W_i)$ and $T_O = T(O_i)$.*

If there does not exist a delay such that the delay-transition pair is enabled in the first state, and firing it leads to the second state, then the transition is semimodular because it cannot fire from the first state. If there does exist a delay such that the delay-transition pair is enabled and firing it from the first state results in the second state, then the transition must be semimodular for correctness to hold. This can be dynamically validated as a firing sequence is evolved. This alone is not sufficient to validate the output semimodular property, the reachable state space must be examined in the end to check for other output semimodular failures; thus, both the reachable state set and allowed firing sequences are required to validate it. This check alone, however, does detect most, if not all, output semimodular failures.

3.2.4 Constraint Satisfied

The level-ruled Petri net model of a component in a system of nets is constraint satisfied if it does not violate its timing and ordering specification. If a designer needs to check a bounded response property in the component, then the component must satisfy the bounded response property to be correct. Similarly, if the designer needs to check that signal transitions in the component are ordered with other transitions, then the component must satisfy the orders to be correct.

The constraint satisfied property is checked through constraint rules. These must be satisfied at the firing of delay-transition pairs.

Definition 3.16 (Constraint Satisfied Delay-transition pair). *A given delay-transition pair (d, t) is constraint satisfied in the state s if $s \vdash (d, C(t))$; recall that $\vdash$ is conjunctive for a left-hand tuple argument such as $s = (\mu, \nu, \mathcal{C})$, and $\nvdash$ is* *disjunctive for the same argument.*

Definition 3.16 requires all constraint rules for a transition to be marking, level, and time satisfied. Recall that from Definition 2.29 that $(\mu, \nu, \mathcal{C}) \vdash (d, R(t))$ implies that the delay d is enabled in $(\mu, \nu, \mathcal{C})$ and the rule set $R(t)$ is satisfied by the new state $(\mu, \nu, \mathcal{C} + d)$ where time has advanced by d. The rule set may not be satisfied in the current state without time first advancing. This is the same for a constraint rule set. All clocks on constraint rules for the transition must be below their upper bound in the clock assignment function after time is advanced by the delay d for the transition to be time satisfied by Definition 3.6. A delay-transition pair that is not enabled in the state may or may not be constraint satisfied. If it is possible for a clock on a constraint rule to exceed its upper bound while waiting for the delay-transition pair to become enabled, then the delay-transition pair is not satisfied.

Definition 3.17 (Constraint Satisfied Firing Sequence). *A firing sequence* $(\mathbf{d}, \mathbf{t})$ *is constraint satisfied for a given set of transitions* $T' \subseteq T$ *if in its corresponding state vector* $\mathbf{s}$ *the following holds for* $1 \leq i \leq [\mathbf{t}]$*:*

1. $t_i \in T' \implies (d_i, t_i)$ *is a constraint satisfied delay-transition pair given* s_{i-1}*; and*

2. *for all transitions* $t' \in T'$ *and for all rules* $r \in C(t')$*,* $(\mu_i, \nu_i) \vdash \{r\} \implies \mathcal{C}_i(r) \leq \mathsf{Lft}(r)$*.*

The constraint satisfied property is checked in a firing sequence of the network considering only transitions related to a component by Definition 3.17. The set T' identifies transitions of interest. In firing a transition from T' on the sequence, the first condition in Definition 3.17 requires the constraint rules for the transition to be satisfied by the associated state from which it fires with its associated delay. The second condition, however, applies to every step of the firing sequence. At any point in the firing sequence, all constraint rules associated with transitions in T' that are marking and level satisfied in the current state must have clock valuations in the same state under their defined latest firing times.

Condition two in Definition 3.17 ensures that constraint rules never violate their latest firing time regardless of the enabling condition of their associated transition. Consider the level-ruled Petri net fragment in Fig. 3.8 with a current state $s = (\mu, \nu, \mathcal{C})$ such that $\mu \vdash R(t_2)$, $\nu \vdash (R(t_1), R(t_2))$, $\mathcal{C}(r_1) = 3$, and $\mathcal{C}(r_2) = 4$. Transition t_1 never fires in this example since its place is not marked. If the delay-transition pair $(2, t_2)$ fires in the given state with condition two omitted from Definition 3.17, then the clock valuation of r_1 in the new state is not a violation of the constraint satisfied property even though its new clock valuation is 5. Every constraint rule pertinent to a component must be checked at every step of a firing sequence because its associated transition may never fire to force the check. The latest firing time must always hold in all marking and level satisfied constraints of interest in every state of the firing sequence for the sequence to be constraint satisfied.

The definition of constraint satisfied firing sequences leads naturally to the constraint satisfied property for a component in a larger system.

Definition 3.18 (Constraint Satisfied). *A component M_i in a network of level-ruled Petri nets $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is constraint satisfied if for all firing sequences $(\mathbf{d}, \mathbf{t}) \in \mathcal{P}(s_o)$ of M, $(\mathbf{d}, \mathbf{t})$ is constraint satisfied when given the set of transitions T_i from the component M_i.*

Section 3.1 presents two examples that use constraint rules to check time separations and firing orders. Consider the first example, the STARI FIFO, shown in Fig. 3.1. The first stage model for the STARI FIFO now contains a constraint rule as shown in Fig. 3.2(a). The constraint rules check that *ack1+* fires before new data is

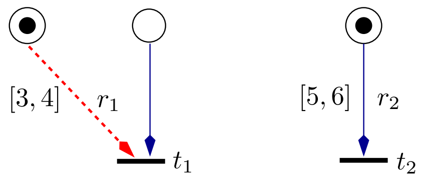


Fig. 3.8. A net fragment that is not constraint satisfied.

inserted into the FIFO; thus, each data value output by the transmitter is inserted into the FIFO before the transmitter sends another data value. Consider now the level-ruled Petri net model for the entire two-stage STARI FIFO with constraint rules in Fig. 3.9. The constraint rules for *x0.t+* and *x0.f+* have infinite latest firing times; thus, the second condition of Definition 3.17 is of no importance for this example. If the earliest firing times are raised to be 14 rather than 3, then the first stage model is not constraint satisfied because *clk+* fires at time 12 and either *x0.t+* or *x0.f+* fire at a time in the 12 to 13 range. The earliest firing time of either of the constraint rules is not time satisfied with a latest firing time of 14 instead of 3.

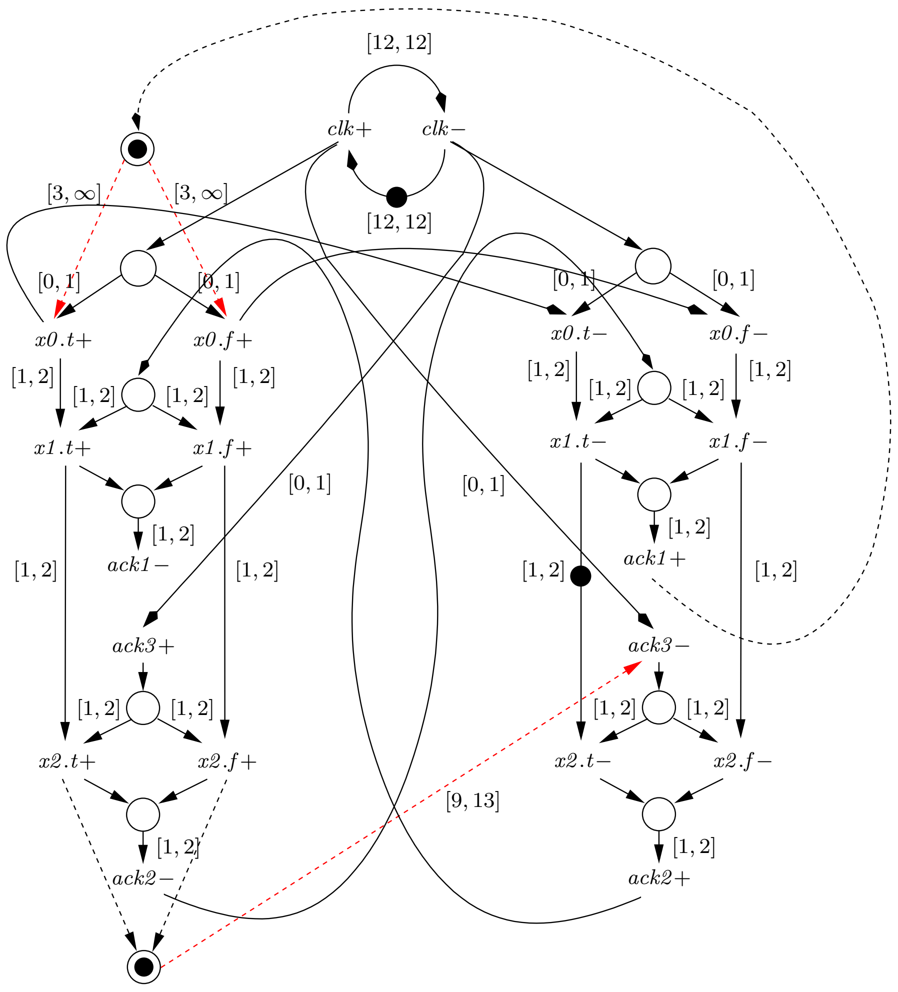


Fig. 3.9. The level-ruled Petri net for the STARI FIFO with constraints.

Consider now the constraint rule in Fig. 3.2(b) for the second stage of the FIFO. The timing requirement on the constraint rule for *ack3*$-$ sets an earliest and latest bound on the time difference between *x2.t*$+$ or *x2.f*$+$ and *ack3*$-$; thus, a new data value is output by the FIFO before each acknowledgment from the receiver. Consider the complete level-ruled Petri net with the constraints for the two-stage STARI FIFO in Fig. 3.9. If the earliest firing time for the constraint rule on *ack3*$-$ is set higher than 9, then the first stage model is not constraint satisfied. To see this failure, consider the case where *clk*$-$ just fired and the state is such that the constraint rule for *ack3*$-$ is not marking satisfied. The *ack3*$+$ transition fires as late as 1 time unit after the *clk*$-$ transition. Either *x2.t*$+$ or *x2.f*$+$ can fire as late as 2 units after *ack3*$+$. The constraint rule for *ack3*$-$ cannot be marking satisfied any earlier than 3 time units after *clk*$-$. There is a 12 time unit separation between *clk*$-$ and *clk*$+$, and there is a minimum 0 unit delay between *clk*$+$ and *ack3*$-$. If the earliest firing time of the constraint rule for *ack3*$-$ is raised to 10, then stage one is not constraint satisfied because *ack1*$-$ can fire when the clock valuation on its constraint rule is only 9. If the latest firing time is set lower than 13, then the component is not constraint satisfied either. In this case, from the shown initial marking, *clk*$+$ fires at time 12 and *ack3*$-$ can fire 1 time unit later. The result is that the timer on the constraint rule can have a value as large as 13 when *ack3*$-$ fires.

Section 3.1 demonstrates the use of constraint rules in a model with syntactic abstraction. The constraint is added in the second stage model of a delayed-reset domino gate computing $f_2 = f_1 \wedge c$. The delayed-reset domino gate with the model and its constraint rule is shown in Fig. 3.3. In this example, the timer for the constraint rule is not started as soon as it is marking satisfied, but when it is level satisfied too. Consider the situation where *clk2*, f_1, and c are high. If the transition $c-$ fires 26 time units after *clk2*$+$, then the *clk2* model is not constraint satisfied because *clk2*$-$ violates the earliest firing time of its constraint rule. Its clock valuation is only 4 when *clk2*$-$ fires.

3.3 Correctness Statement

This section formalizes the correctness statement and completes the definition of the reachable states and firing sequences for a level-ruled Petri net. The current definitions are incomplete because they do not include any notion of failure. As noted previously, the behavior of a level-ruled Petri net in the presence of some failures—unsafe or inconsistent state assignment—is not defined.

Definition 3.19 (Module Correctness). *A component M_i in a network of level-ruled Petri nets $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is correct if it is safe, consistent state assigned, output semimodular, and constraint satisfied in the parallel composition of the network M; the violation of the correctness property is a failure.*

The correctness condition of a component is validated by exploring all allowed firing sequences and states in the parallel composition of the network.

Definition 3.20 (Unbounded Failure Behavior). *The behavior of a component M_i in a network of level-ruled Petri nets $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is not defined from the point of a failure; if a firing sequence of the parallel composition of the network contains a failure for the component, then any transition is possible after that point.*

Definition 3.20 allows a firing sequence containing a failure to always be a failure from the failure point on. Its behavior is completely unbounded after the failure.

The next definition in this chapter covers a complete valid network of level-ruled Petri nets. A notion of *pure environment model* must be presented for the definition. In a network of level-ruled Petri nets, not all nets refer to a physical component. Some nets exist solely to generate random input or mimic an interface. These components are pure environment modules and are represented by $\mathcal{E}$ in the parallel composition. The environment $\mathcal{E}$ is a level-ruled Petri net but is not intended to be implemented in the system.

Definition 3.21 (System Correctness). *A network of level-ruled Petri nets $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is correct if all of its modules M_i for $1 \leq i \leq n$ are correct* *in the parallel composition of the network M, and its environment module $\mathcal{E}$ is safe, consistent state assigned, and constraint satisfied in the same parallel composition M; the violation of the correctness property is a failure.*

Every module is correct in a correct system where correctness is given by Definition 3.19. The environment module, however, does not fall under Definition 3.19. As mentioned previously, a pure environment module is usually not output semimodular because it often generates random outputs; thus, this property is not required. All of the other correctness properties, however, must hold.

The final definition for this chapter is useful to timing analysis. It defines a function to return failures as transitions are fired in state space exploration. The function, however, only addresses untimed failures, or failures that do not involve time.

Definition 3.22 (Untimed Failure). *The function* $\mathsf{untimed\_failure}(\mu, \nu, t, \mu', \nu')$ *returns a failure if there exists a module M_i in $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$ such that any of the following hold:*

1. ***Safety Failure**: t is not a safe transition in μ given P by Definition 3.7;*
2. ***Consistent State Assignment Failure**: t is not a consistent transition in (μ, ν) by Definition 3.10;*
3. ***Output Semimodular Failure**: $t \in T_i$, and it is not a semimodular transition in (μ, ν) and (μ', ν') for $T_V = T(W_i)$ and $T_O = T(O_i)$ by Definition 3.13; or*
4. ***Constraint Failure**: $t \in T_i$ and $(\mu, \nu) \nvdash C(t)$ by Definition 3.16.*

Notice that there is an untimed portion to the constraint satisfied property. This is validated in the untimed failure function. Another important point is that the scope of the check is not limited in the safe and consistent state assignment failures. These failures are always reported regardless of the module they occur in. The reduction method in Chapter 5 may find failures outside of the target module, but it guarantees to find failures in the submodule only if they exist.

3.4 Related Work

In formal methods, model checking is appealing because is allows the verification of general properties. A model is often specified in a temporal logic such as LTL in [45] or CTL in [46, 47, 48], and it models behaviors that the circuit should uphold. To specify timed behaviors, it is necessary to use a more complex and expressive timed temporal logic such as those presented in [49, 50, 51, 18, 52]. With any temporal logic, the circuit is framed in an underlying representation that captures essential behaviors in the circuit. A circuit is then shown to satisfy a model by exploring all behaviors allowed in the underlying representation. If a behavior is allowed in the circuit but not captured by the internal representation, or the desired behavior is not specified in the model, then it is not checked in the verification process; thus, correctness in model checking is a statement that the representation of the circuit satisfies behaviors specified in the model.

Trace theory is less expressive than general model checking, but it does formalize conformance of a circuit to its specification. Showing conformance in trace theory is easier than showing that an arbitrary model is satisfied by a circuit. In Dill's canonical trace structures [53], a failure occurs when an output is generated by the circuit, and the mirror of its specification cannot accept it as an input; or conversely, when the mirror of the specification produces an output, the circuit is not in a state to accept it as an input. The use of the mirroring theorem allows for efficient verification of safety properties. General liveness properties can be checked with complete trace structures in [53], but the complexity of the verification problem is extremely prohibitive.

Various extensions have been made to trace theoretic verification. In [54], the idea of strong conformance is introduced to verify some notion of liveness, as well as safety properties in asynchronous circuits. A notion of time is introduced in [55] by extending the trace alphabet with time symbols. A time symbol denotes a discrete time step; and it is shown in a large class of circuits that conformance in discrete time implies conformance in continuous time. Work in [55] includes discrete time bounded response in the correctness definition. In [56], continuous time is integrated into trace theory and applied to time Petri nets. A time Petri net has delay bounds associated with each transition in the net. Trace theory is extended to Orbital nets in [57]. Orbital nets are similar to timed Petri nets and differ from time Petri nets in [56] in that places, not transitions, are annotated with delay information. Work in [57] extends correctness in trace theory to include continuous time bounded response in the environment. Recent work in [58] relaxes requirements in [57] on the environment. Its correctness definition shows timed conformance of an implementation to a specification. Timed conformance can be verified using constraint rules in a manner similar to [58]. The specification rules become constraint rules on the implementation.

The work that most closely resembles this definition of correctness is that done by Belluomini in [43]. She introduces the constraint rule in timed event/level structures and a property similar to the constraint satisfied property defined here. She does not define, however, the safe, consistent state assignment, and output semimodular properties. She also uses trace theory as the theoretical basis for the correctness properties.

3.5 Summary

This chapter presents a formal definition of the correctness of a component in a larger system. To support the correctness definition, this chapter presents the constraint rule. Timing and ordering properties are specified using constraints rules. This chapter refines the level-ruled Petri net semantics to deal with constraint rules. A constraint rule is essentially an observer in the net whose clock is reset when appropriate, but never used to affect the firing of any transition.

This chapter defines four correctness properties for a component in a network of level-ruled Petri nets. The first three properties, safe, consistent state assigned, and output semimodular are common correctness requirements in verification models. The last property, constraint satisfied, is less common. The constraint rules in a component are checked in each transition firing to ensure than none of the timers on the rules exceed their latest firing time. Whenever a transition in the component fires, however, the transition's constraint rules are checked to see that they are satisfied in the current state of the system. This enables a designer to specify bounded response and ordering properties in noncausally related transitions.

The chapter formally presents a correctness statement and defines an untimed failure function that returns an untimed failure. An untimed failure relates to safety, consistent state assignment, and output semimodular violations. It also covers the case where a transitions fires with a constraint rule that is not satisfied by the current marking. The function can be used by timing analysis to check the untimed portion of correctness in a level-ruled Petri net model of a timed circuit. A function to check for timed failures is given in Section 4.7. It is important to be able to check that timers on constraint rules do not exceed their upper bounds. The operations required for this are best presented in Chapter 4 with the formal definition presented here.

CHAPTER 4

TIMING ANALYSIS

A circuit is verified correct by searching for failures in its allowed firing sequences and reachable states for a given environment. The allowed firing sequences and set of reachable states is typically infinite for the level-ruled Petri net. To perform the analysis, a finite representation of the infinite space must be realized.

A finite representation of the firing sequences and reachable states exists. The two sets can be represented by a finite graph. Each node of the graph is an equivalence class, representing a set of timed states. The edges are transitions between equivalence classes. The paths through the graph are the allowed firing sequences. Each reachable timed state in the system maps into an equivalence class in the graph. Each allowed firing sequence of the system maps into a path of the graph. This creates a finite graph representation.

The size of the graph representation depends on the size of the equivalence classes at each node. If the equivalence classes are small, then many are required to represent the complete reachable state space. If the classes are too large, however, then extra behaviors may be represented that are not part of the reachable state space. Care must be taken in building equivalence classes.

This chapter presents an algorithm to construct a finite representation of the reachable states and firing sequences in a level-ruled Petri net. The algorithm implements a partial order in the timing information to reduce the size of the finite representation. The chapter is structured to mitigate the complexity of the algorithm. Section 4.1 presents the finite representation of the infinite system. It presents the timed state class, and a detailed discussion of the zone, which is a key component in the timed state class.

Section 4.2 presents the causal group. This is an important contribution of this research. It facilitates the analysis of arbitrary Boolean expressions on rules. It also enables the analysis algorithm to directly compute causality. This improves running time performance because it does not need to explore redundant firing orders on rules.

The timing analysis algorithm is divided into 4 distinct parts: computing fireable transitions, creating successor states, building the finite state space, and pruning transitions from the timing representation. Section 4.3 presents the algorithm to compute fireable transitions in a state. It is equivalent to the enabled property in the level-ruled Petri net. Section 4.4 is the algorithm to create successor states after firing a transition. This is equivalent to the update definitions for the marking, Boolean state, and clock assignment function. This is where the partial order in the timing information is implemented. It does not order transitions to fire after other already fired transitions. This effectively reduces the size of the finite representation. Section 4.5 is the algorithm that builds the finite representation of the state space. The algorithm is a depth first search using the fireable and successor algorithms. Section 4.6, finally, is an algorithm to remove redundant transitions from the timing information. It is key in reducing the size of the final timed state representation.

Section 4.7 adds to the algorithm support to validate timing correctness. The earliest and latest firing time failures are presented along with an algorithm to validate them. Section 4.8 discusses issues in removing transitions in constraint rules from the time state class representation. Section 4.9 addresses the issue of exactness in the finite representation. The timing analysis algorithm approximates timed dependent choice behavior. This makes the resulting finite representation conservative in that it can include states that are not reachable in the level-ruled Petri net. This chapter is concluded in two more sections. Section 4.10 is a discussion of related timing analysis algorithms. Section 4.11, finally, concludes this chapter with a brief summary of its contributions.

The presentation in this chapter assumes the level-ruled Petri net that is being analyzed is global. Any reference to the set of transitions T, rules R, etc. refer to the globally known level-ruled Petri net. The reader is encouraged to be patient in this section. Although measures are taken to present the material in a concise and logical pattern, it is a complex algorithm that requires several definitions to present. The end goal of this chapter is to completely define the analysis algorithm to a point where it can be recreated from this text.

4.1 Equivalence Classes

There are two critical points of understanding formalized in this section. The first is the zone as the cornerstone of the equivalence class. Although the zone presentation is detailed, the salient point is that the zone sets bounds on the minimum and maximum separation between transitions. These bounds are not just for transitions that are directly related by causality, but all transitions in the zone. The second critical point of understanding in this section is the timed state class. It replaces the clock assignment function in the timed state with the zone. The salient point is that the timed state class is a replacement of the timed state. Although not defined in this section, a timed state replacement must have a notion of enabled delays, enabled transitions, and how to fire these things to move the system to a new timed state class. The goal in this section, however, is not to define these terms, but to present the timed state class as a timed state replacement and build the foundation for how the timed state class is to be used to construct a finite representation of the infinite state space of the level-ruled Petri net. The foundation is the idea that the rules for a transition implicitly define separations between the transition and other transitions involved in causing the rule to be marking and level satisfied. This is the driving idea behind the zone and the timed state class. The idea is formalized in the next section.

This section begins with a detailed presentation of the zone, including its interpretation, structure, and a set of defined operations. These operations provide an interface to algorithms. The timed state class is defined after the zone presentation. The presentation is largely algorithmic. The definitions that are presented in this section are constructive, meaning that algorithms to implement the definitions are readily available or presented in the text.

4.1.1 Zones

The zone is key to creating a finite representation for the infinite state space of the level-ruled Petri net. A state of the level-ruled Petri net consists of a marking, Boolean state, and a clock assignment function. The clock assignment function maps rules in the level-ruled Petri net to real values representing clocks. The clock assignment function makes the state space of the level-ruled Petri net infinite. The clock assignment functions must be gathered into equivalence classes to create a finite representation. The is the role of the zone.

The zone is a convex polygon in n-dimensional space, where n can be any natural number. The polygon is captured in a square matrix.

Definition 4.1 (Square Matrix). *A square matrix A of size n is the array*

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix};$$

where each element is either an integer number or the symbol ∞, representing infinity; the function $[A]$ returns the dimension of the matrix A.

Each a_{ij} entry in the matrix is the inequality: $\tau(t_i) - \tau(t_j) \leq a_{ij}$. The function $\tau(t)$ returns the firing time transition t. A matrix entry thus defines a range on the difference between the firing time of t_i and t_j. In a similar fashion, the entry a_{ji} is the inequality: $\tau(t_j) - \tau(t_i) \leq a_{ji}$. Recall that $\tau(t_j) - \tau(t_i) \leq a_{ji}$ is equivalent to $\tau(t_i) - \tau(t_j) \geq -a_{ji}$. Combining the two inequalities now creates a minimum and maximum difference that can exist on the firing times of t_i and t_j: $-a_{ji} \leq \tau(t_i) - \tau(t_j) \leq a_{ij}$. For the remainder of this presentation, the call to the time function τ is implied; thus, $t_i - t_j \leq a_{ij}$ is equivalent to $\tau(t_i) - \tau(t_j) \leq a_{ij}$.

Recall from Chapter 2 that a vector is an ordered sequence of elements $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Its length is denoted by $[\mathbf{x}]$. The index function is an operation on vectors of any type and a necessary function for the zone presentation.

Definition 4.2 (Index Function). *The function* $\mathsf{ind}(\mathbf{x})(x) = \{i \in \mathbb{N} \mid x_i = x\}$ *returns the set of indices to the vector* $\mathbf{x}$ *where* x *appears.*

The index function is used to find indices for entries in the zone and to track the number of instances of a given element in the zone.

Definition 4.3 (Zone). *A zone is the tuple* $z = (\mathbf{t}, A)$ *consisting of a vector of transitions* $\mathbf{t}$ *and a square matrix* A *where* $[\mathbf{t}] = [A]$ *and for all* $t \in T$, $|\mathsf{ind}(\mathbf{t})(t)| \leq 2$*; for any pair of natural numbers* $(i, j) \in \mathbb{N}$ *such that* $1 \leq (i, j) \leq [\mathbf{t}]$*, the entry* a_{ij} *in the matrix* A *is the maximum amount of time that can elapse from the firing of* t_j *to the firing of* t_i *in the vector* $\mathbf{t}$*.*

The square matrix is a polygon that defines the allowed time separations between transitions in the transition vector. A transition entry in a zone is an instance of the transition. The zone does not allow more than two instances of any transition. For a transition t with two instances in the zone, $\min(\mathsf{ind}(\mathbf{t})(t))$ is the index to the first or oldest instance of the transitions, and $\max(\mathsf{ind}(\mathbf{t})(t))$ is the index to the last or most recent instance of the transition in the zone. A zone must always have at least one entry in its transition vector.

The maximum amount of time that can elapse from the firing of t_j till the firing of t_i from the vector $\mathbf{t}$ is less than or equal to a_{ij}. Similarly, the maximum amount of time that can elapse from the firing of t_i till the firing of t_j is less than or equal to a_{ji}. Note that either value can be negative indicating an order on the two transitions. This relation creates a range on the separation of the two transitions. An understanding of the meaning of this inequality is key to the algorithm. Consider the zone in Fig. 4.1(a). The transition vector map is the first row of the zone, and it is shown again in the first column for convenience. The zone states that the maximum amount of time that can elapse from the firing t_2 till the firing of t_1 is -3; thus, the firing or t_1 always precedes the firing of t_2 by at least 3 time units. To state it differently, the firing time of t_2 always follows the firing time of t_1 by at least 3 time units. Similarly, the zone states that the maximum amount of time that can elapse from the firing of t_1 till the firing of t_2 is 7 time units; thus, the firing time of t_2 always follows the firing time of t_1 by at most 7 time units. The two inequalities combined are given as: $3 \leq t_2 - t_1 \leq 7$. This inequality can be a useful perspective. An a_{ij} entry can be thought of as the time at which t_i fires minus the time at which t_j fires. That separation must be less than or equal to a_{ij}; thus, the time of the firing of t_2 minus the time of the firing of t_1 must be less than or equal to 7, and the time of the firing of t_1 minus the time of the firing of t_2 must be less than or equal to -3. The zone in Fig. 4.1(a) also defines the range $2 \leq t_3 - t_1 \leq 5$ for transitions t_3 and t_1. Notice that the separations for t_2 and t_3 are unconstrained in this zone as indicated by ∞.

There are many zones that represent the same polygon. Zones cannot be used to build a finite representation of the state space unless they have a canonical form. The canonical form enables the comparison of any two arbitrary zones. Every zone has a canonical form, and identical zones have identical canonical forms.

Definition 4.4 (Canonical Form). *A zone $z = (\mathbf{t}, A)$ is canonical if it is maximally constrained; $\check{z}$ is the canonical form of the zone z.*

All inequalities in the zone are maximally tight in the canonical form. A zone is put in its canonical form by applying Floyd's all pairs shortest path algorithm to the matrix A treated as a dense graph where for all $(i, j) \in \mathbb{N}$ such that $1 \leq (i, j) \leq [\mathbf{t}]$, the entry a_{ij} is the weight of the edge between nodes i and j. Consider now the zone in Fig. 4.1(b). This is the canonical form of the zone in Fig. 4.1(a). The separation of 2 between t_3 and t_2 is from the fact that t_3 goes to t_1 with a weight of 5 and t_1 goes to t_2 with a weight of -3; thus, the edge between t_1 and t_3 is constrained to 2. The entry for t_2 and t_3 is found in a similar fashion. The canonical form of the zone indicates that t_2 can fire no later than 5 time units after t_3; and similarly, t_3 can fire no later than 2 time units after t_2. From this, the two transitions are concurrent in that they can fire in any order.

Not all zones have a logical meaning. An entry in the zone is a maximum time separation between the firing of one transitions to the firing of another transition. A separation between a transition and itself must be zero or the zone does not make sense.

Definition 4.5 (Consistent Zone). *A zone* $z = (\mathbf{t}, A)$ *is consistent if for all* $i \in \mathbb{N}$ *such that* $1 \leq i \leq [\mathbf{t}]$, $a_{ii} = 0$.

In the case of a rule where a transition can mark its own place, then two instances of the transition are entered into the zone; thus, a nonzero separation exists between the two different instances of the transition, and a zero separation exists between the same instance of the transition in the zone. Only zones that are consistent with there meaning are used to build a finite representation of the state space.

The order in which transitions appear in the zone reflect an implicit firing order. This is an important property to building a finite representation because it helps detect invalid firing orders. Moving from the first index to the last index in the transition vector in the zone, the transition at the current index must not be ordered to strictly fire before transitions at earlier indices. It must be allowed to fire either after or concurrently with transitions at earlier indices.

Definition 4.6 (Valid Zone). *A zone* $z = (\mathbf{t}, A)$ *is valid if it is consistent and for all* $(i, j) \in \mathbb{N}$ *such that* $1 \leq i \leq [\mathbf{t}]$ *and* $j < i$, $a_{ij} \geq 0$.

For an index i and an index $j < i$, the maximum time that can elapse from the time of the firing of t_j and the time of the firing of t_i must be positive as denoted by $a_{ij} \geq 0$. If $a_{ij} < 0$, then t_j fires at a time later than t_i; thus, t_i is out of order with respect to t_j in the zone because it must always fire before earlier entries in the zone. The upper triangle of the matrix is not considered because the zone reflects maximum separations. A valid zone allows t_i to fire after t_j in the maximum for $i < j$. That is not to say that t_j cannot fire after t_i in the maximum too. This is shown in Fig. 4.1(b). This zone is valid because it preserves the implicit order on transition entries. The maximum amount of time that can elapse between the firing of t_2 to the firing of t_3 is 2; thus, t_3 can fire after t_2 in the maximum. Now consider the maximum amount of time that can elapse between the firing of t_3 and the firing of t_2. This separation is 5. Transition t_2 can fire after t_3 in the maximum and appears before t_3 in the zone, but the zone is still valid because the other order is allowed by the zone too.

Relational operators between zones are necessary to construct the finite representation of the infinite state space. The relations help exclude duplicate zones and zones that are smaller than other zones in the state space. All of the necessary relations are constructed from an equality and superset relation.

Definition 4.7 (Equality Relation). *The zone $z = (\mathbf{t}, A)$ is equal to the zone $z' = (\mathbf{t}', A')$ if $[\mathbf{t}] = [\mathbf{t}']$ and for all $(i, j) \in \mathbb{N}$ such that $1 \leq (i, j) \leq [\mathbf{t}]$, there exists $(k, l) \in \mathbb{N}$ such that, $1 \leq (k, l) \leq [\mathbf{t}']$ where $t_i = t'_k$ in $\mathbf{t}$ and $\mathbf{t}'$, $t_j = t'_l$ in $\mathbf{t}$ and $\mathbf{t}'$, $a_{ij} = a'_{kl}$ in A and A'; and for all $(k, l) \in \mathbb{N}$ such that $1 \leq (k, l) \leq [\mathbf{t}]$, there exists $(i, j) \in \mathbb{N}$ such that, $1 \leq (i, j) \leq [\mathbf{t}']$ where $t_i = t'_k$ in $\mathbf{t}$ and $\mathbf{t}'$, $t_j = t'_l$ in $\mathbf{t}$ and $\mathbf{t}'$, $a_{ij} = a'_{kl}$ in A and A'; the relation is indicated by $z = z'$.*

Equal zones are equal in every sense of the word but order in the transition vectors. Although two zones may show a different order on transitions in their vector mappings, if the separations are identical for all transition pairs, then the two zones are equal. This implies that different transition firing orders lead to identical zones, since the order of entry in the zone implies the order of firing by the valid property. This is an important characteristic to be exploited in timing analysis.

Definition 4.8 (Superset Relation). *The zone $z = (\mathbf{t}, A)$ is a superset of the zone $z' = (\mathbf{t}', A')$ if $z \neq z'$, $[\mathbf{t}] \leq [\mathbf{t}']$, and for all $(i, j) \in \mathbb{N}$ such that $i \neq j$ and $1 \leq (i, j) \leq [\mathbf{t}]$, there exists $(k, l) \in \mathbb{N}$ such that $k \neq l$, $1 \leq (k, l) \leq [\mathbf{t}']$ where $t_i = t'_k$ in $\mathbf{t}$ and $\mathbf{t}'$, $t_j = t'_l$ in $\mathbf{t}$ and $\mathbf{t}'$, $a_{ij} \geq a'_{kl}$ in A and A', and if $i = \min(\mathsf{ind}(\mathbf{t})(t_i))$ then $k = \min(\mathsf{ind}(\mathbf{t}')(t_k))$; the relation is indicated by $z \supset z'$; the other relational operators ($\subset$, $\subseteq$, $\supseteq$, etc.) are defined appropriately using the superset and equality relations.*

Like the equality relation, the superset relation does not need to match on the order of transitions in the two zones. The superset relation, unlike the equality relation, however, makes use of missing entries in the zone. Consider the zone in Fig. 4.1(c).

$$\begin{array}{c|ccc} & t_1 & t_2 & t_3 \\ \hline t_1 & 0 & -3 & -2 \\ t_2 & 7 & 0 & \infty \\ t_3 & 5 & \infty & 0 \end{array}$$

(a)

$$\begin{array}{c|ccc} & t_1 & t_2 & t_3 \\ \hline t_1 & 0 & -3 & -2 \\ t_2 & 7 & 0 & 5 \\ t_3 & 5 & 2 & 0 \end{array}$$

(b)

$$\begin{array}{c|cc} & t_2 & t_3 \\ \hline t_2 & 0 & 5 \\ t_3 & 2 & 0 \end{array}$$

(c)

Fig. 4.1. A zone, its canonical form, and a superset relation. (a) A zone with three dimensions. (b) The canonical form of the zone. (c) A zone that is a superset of the zone in (b).

This zone is a superset of the zone in Fig. 4.1(b). Although it has a different dimensionality, it agrees on each separation for transitions common to the two zones. The missing dimension is considered unconstrained in the superset relation. A zone of smaller dimension can be a superset of a zone of larger dimension because the missing dimensions are completely unconstrained relative to entries in the zone. The zone in Fig. 4.1(c) is not a superset of the zone in Fig. 4.1(a), however, because the defined separations for common transitions in the zone of Fig. 4.1(a) are larger than those in zone Fig. 4.1(c).

A zone is constructed one transition at a time. The ability to add new transitions into the zone is a basic operation in building zones.

Definition 4.9 (Adding Transitions). *The addition of a transition $t \in T$ to the zone $z = (\mathbf{t}, A)$ creates a new zone $z' = (\mathbf{t}', A')$ if $|\mathsf{ind}(\mathbf{t})(t)| < 2$, where $\mathbf{t}'$ is the result of the concatenation of t to the end of $\mathbf{t}$, $[A'] = [\mathbf{t}']$, and for all $(i, j) \in \mathbb{N}$ where $1 \leq (i, j) \leq [\mathbf{t}']$,*

$$a'_{ij} = \begin{cases} 0 & \textit{if } i = j = [\mathbf{t}'], \\ \infty & \textit{if } (i = [\mathbf{t}'] \vee j = [\mathbf{t}']) \wedge i \neq j, \textit{ and} \\ a_{ij} & \text{otherwise.} \end{cases}$$

An added transition is always placed at the end of the transition vector in the zone. As mentioned previously, the order of transitions in the zone reflects a relative firing order. Adding the transition to the zone denotes the firing of the added transition. The separations in the matrix relating the new transition and the old transitions are left unbounded. Although they must be set to reflect firing separations allowed by the level-ruled Petri net, the discussion on how these separations are created is a topic for Section 4.2. This section simply provides a necessary interface.

Transitions can be deleted, as well as added, to a zone. Unlike adding a transition, where the new transition is added to the end of the map and matrix, any transition can be deleted from anywhere in the map and matrix.

Definition 4.10 (Vector Delete). *The delete function* $\mathsf{del}\,(\mathbf{x})\,(i)$ *returns the vector of elements in* $\mathbf{x}$*, but not including, the element at index* $i \in \mathbb{N}$ *and is defined as*

$$\mathsf{del}\,(\mathbf{x})\,(i) = \begin{cases} \mathbf{x} & \textit{if } i > [\mathbf{x}] \textit{ and} \\ (x_1, \ldots, x_{i-1}).(x_{i+1}, \ldots, x_n) & \text{otherwise;} \end{cases}$$

where the '.' operation is the concatenation of the two vectors.

Vector delete returns a new vector that includes everything but the entry at transition i.

It is harder to delete a transition from the zone than it is to add a transition to the zone. This is due to the fact that transitions are ordered in the zone, and deletion is not restricted to the first or last entry of the zone.

Definition 4.11 (Deleting Transitions). *The deletion of a transition* $t \in T$ *from the zone* $z = (\mathbf{t}, A)$ *creates a new zone* $z' = (\mathbf{t}', A')$ *if* $|\mathsf{ind}(\mathbf{t})(t)| \geq 1$ *where* $i = \min(\mathsf{ind}(\mathbf{t})(t))$, $\mathbf{t}' = \mathsf{del}\,(\mathbf{t})\,(i)$, $[A'] = [\mathbf{t}']$, *and for all* $(j, k, l, m) \in \mathbb{N}$*:*

$$a'_{jk} = \begin{cases} a_{jk} & \textit{if } j < i \wedge k < i, \\ a_{jm} & \textit{if } j < i \wedge k \geq i, \\ a_{lk} & \textit{if } j \geq i \wedge k < i, \textit{ and} \\ a_{lm} & \textit{if } j \geq i \wedge k \geq i. \end{cases}$$

where $1 \leq (j, k) \leq [\mathbf{t}']$, $l = j + 1$, *and* $m = k + 1$.

Notice that in this definition, it is always the first instance of the transition that is deleted. This transition relates to the oldest instance of a transition that appears multiple times. At an intuitive level, deleting a transition removes its corresponding row and column from the zone matrix. The four cases in the definition simply detect when one of the current indices refers to the index of the deleted transition and then jumps over the entries for the deleted transition.

The timing analysis algorithm needs to look at transition separations existing in the zone. A transition, however, can appear as many as two times in the zone. A transition in the zone is an instance of the firing of that transition. In building a zone, it is important to distinguish which instance of a transition is being referenced. The next two definitions clarify this point for setting and observing separations in the zone.

Definition 4.12 (Min-max Entry). *A min-max entry in a zone $z = (\mathbf{t}, A)$ for a transition pair (t_i, t_j) is a_{ij} where $i = \min(\mathsf{ind}(\mathbf{t})(t_i))$ and $j = \max(\mathsf{ind}(\mathbf{t})(t_j))$ if $|\mathsf{ind}(\mathbf{t})(t_i)| > 0$ and $|\mathsf{ind}(\mathbf{t})(t_j)| > 0$; otherwise it does exist.*

The min-max entry gives the separation on the oldest instance of the first transition and the newest instance of the second transition. If a transition does appear twice in the zone, then its min-max entry is the maximum amount of time that its first instance can fire after its second instance.

Definition 4.13 (Max-min Entry). *A max-min entry in a zone $z = (\mathbf{t}, A)$ for a transition pair (t_i, t_j) is a_{ij} where $i = \max(\mathsf{ind}(\mathbf{t})(t_i))$ and $j = \min(\mathsf{ind}(\mathbf{t})(t_j))$ if $|\mathsf{ind}(\mathbf{t})(t_i)| > 0$ and $|\mathsf{ind}(\mathbf{t})(t_j)| > 0$; otherwise it does exist.*

The max-min entry in a zone returns the separation on the newest instance of the first transition and the oldest instance of the second transition. If a transition does appear twice in the zone, then its max-min entry is the maximum amount of time that its second instance can fire after its first instance Although not formally presented, the min-min and max-max entries are similarly defined.

The idea of concurrent transitions is critical to building a finite representation. Concurrency is related to a set of transitions being enabled. This relation is exploited in the next section. A set of transitions is concurrent in a zone if the zone allows the transitions to fire in any order. The following definition computes this set given a zone and set of transitions in the zone.

Definition 4.14 (Fire First Transitions). *The set of fire first transitions in the set T' given the zone $z = (\mathbf{t}, A)$ is the set of transitions $t_i \in T'$ such that $|\mathsf{ind}(\mathbf{t})(t_i)| > 0$ and for all $t_j \in T'$ such that $|\mathsf{ind}(\mathbf{t})(t_j)| > 0$, $a_{ji} \geq 0$ where $j = \max(\mathsf{ind}(\mathbf{t})(t_j))$ and $i = \max(\mathsf{ind}(\mathbf{t})(t_i))$.*

The definition only considers transitions in the set T'. This is a restricted scope as reflected by the use of the definition in the algorithm presentation. Consider the zone in Fig. 4.2 with the set $T' = \{t_3, t_4, t_5, t_6\}$. The definition considers the submatrix formed by the transitions in the set. This is the bottom right division of the zone in Fig. 4.2. A transition is concurrent if its corresponding column in the submatrix is positive in every entry. Consider the column for t_3. Transition t_4, t_5, and t_6 can all fire after t_3 in the zone; thus, t_3 is added to the set of concurrent transitions. Now consider the column for t_6. Although t_5 can fire after it by at most 10 time units, transitions t_1 and t_2 must always precede it by at least 1 time unit; thus, t_6 is not added to the concurrent set. In this case, t_3 and t_4 are the only concurrent transitions. Notice that in this definition, the last instance of each transition in the T' set is always used. In this way, the newest instance of a duplicate transition is considered for concurrency.

4.1.2 Timed State Class

The zone is the basic building block in constructing a finite representation of the infinite state space of the level-ruled Petri net. This section uses the zone to create an equivalence class of timed states—the timed state class. Recall that a

	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	-1	0	0	-12	-2
t_2	3	0	3	3	-9	-1
t_3	1	0	0	1	-11	-1
t_4	1	0	1	0	-11	-1
t_5	12	11	12	12	0	10
t_6	5	2	5	5	-7	0

Fig. 4.2. A zone showing two concurrent transitions from a set of transitions.

level-ruled Petri net is the tuple $M = (N, E, C)$, where $N = (T, P, F, \mu_o)$ is an ordinary Petri net, $E = (W, L, \nu_o, \mathsf{Eft}, \mathsf{Lft}, \mathsf{Lsat})$ is a level-ruled extension for the net N (Definition 2.10), and C is the set of constraint rules for the model. A timed state class is a finite representation of an infinite number of timed states.

Definition 4.15 (Timed State Class). *A timed state class for a level-ruled Petri net M is the three-tuple $s = (\mu, \nu, z)$ where μ and ν are a marking and Boolean state of M; and $z = \check{z}$ is a zone in its canonical form defined over transitions in M and is a set of clock assignments.*

The timed state class is similar to the timed state from Chapter 2. The only difference is that the clock assignment function is replaced by the zone.

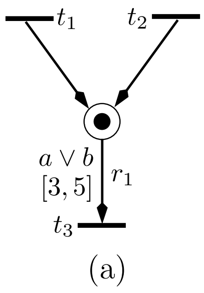
The initial timed state class is given by $s_o = (\mu_o, \nu_o, z_o)$, where μ_o and ν_o are the initial marking and Boolean state from M. The initial zone z_o is derived from μ_o and ν_o. It includes all transitions that must have fired to create the initial marking and Boolean state.

Definition 4.16 (Initial Zone). *The initial zone for a level-ruled Petri net M is the zone with all separations set to zero defined over the set of transition $t \in T$ for which one of the following conditions hold:*

1. *there exists a place $p \in \mu$ such that $t \in \bullet p$; or*
2. *there exists a signal $w \in W$ such that $L(t) = w+$ and $w \in \nu_o$ or $L(t) = w-$ and $w \notin \nu_o$;*

the initial zone is given as z_o.

The first condition includes any transitions that fired to create the marking, and the second condition includes any transitions that fired to create the Boolean state. Consider the fragment of a level-ruled Petri net in Fig. 4.3(a). Suppose that signals a and b are low in the initial Boolean state, $L(t_4) = L(t_5) = a-$, and that $L(t_6) = b-$. The initial zone z_o required by this fragment is given in Fig. 4.3(b). Transitions t_1 and t_2 are included because either transition may have fired to create the initial



$$\begin{array}{c|cccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\ \hline t_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_6 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

(b)

Fig. 4.3. A fragment of a closed system with a portion of its initial zone. (a) The level-ruled Petri net fragment. (b) The initial zone.

marking. Transitions t_4 and t_5 are included because they map to $a-$. Transition t_6 is included because it maps to $b-$.

The level-ruled Petri net semantics are defined relative to a timed state of the system. This works in a timed state class for everything but the zone. The clock assignment function must be approached from a new perspective—time separations on transitions. Recall that a transition is enabled in a timed state if it is marking, level, and time satisfied. The new perspective does not affect marking and level satisfied. Time satisfied, however, cannot be directly checked in a timed state class.

A rule forces a minimum and possibly maximum separation between transitions. Consider again the fragment in Fig. 4.3(a). The r_1 rule implies that the separation between t_3 and either t_1 or t_2 is at least $\mathsf{Eft}(r_1) = 3$ time units. It also implies that this separation must hold for either transition $a+$ or $b+$. The separation is not required on both transitions $a+$ and $b+$ because only one of the two signals must change to level satisfy the expression. Similarly, only t_1 or t_2 fires in a safe level-ruled Petri net, so the $\mathsf{Eft}(r_1)$ is a minimum separation for one of these two transitions. The possible maximum separation is a result of the fact that many different transitions can cause a rule to become either marking or level satisfied; thus, only one of the transitions define the absolute maximum separation $\mathsf{Lft}(r_1) = 5$ between itself and t_3. This can be either the $a+$, $b+$, t_1, or t_2 transition. A causal transition is the last transition to fire to create a state where a rule for a transition is marking or level satisfied. The set of causal transitions can be divided into *causal* *groups.* A causal group is a necessary and sufficient set of transitions to marking and level satisfy another transition. Computing causal groups from the set of causal transitions is the topic of the next section.

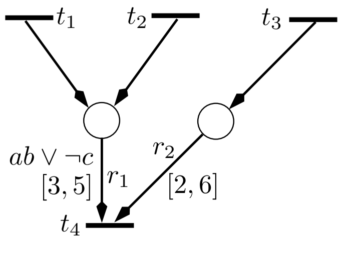
4.2 Causal Group Set

A causal group for a transition is a sufficient set of fired transitions that marking and level satisfy a transition's rule set. The firing time of a transition t must be separated from transitions in one of its causal groups according to the bounds on the rules in $R(t)$. Consider the transition t_4 in the level-ruled Petri net fragment shown in Fig. 4.4. Transition t_4 must not fire earlier than 2 time units after t_3. This is a required separation from the rule r_2. Similar separations exist for the other transitions that mark and level satisfy rules in $R(t_4)$.

Definition 4.17 (Marking Required Set). *The marking required set for a transition $t \in T$ is $\mathsf{mrs}(t) = \bigcup_{p \in \bullet t} \{\bullet p\}$.*

The marking required set is a set of sets. The marking required set for t_4 is $\{\{t_1, t_2\}, \{t_3\}\}$. This falls straight from the structure of the net and is easily computed. The rules in $R(t_4)$ are marking satisfied when a transition from each of these two sets fires: $\{t_1, t_2\}$ and $\{t_3\}$. Note that the definition allows for transitions with a places that contains empty presets. In these cases, the marking required set contains member sets that contain only the empty set.

A similar set of transition sets to level satisfy transition t_4 is less obvious. Syntactic abstraction does not come for free. Boolean functions implicitly create



$L(t_5) = a+$
$L(t_6) = b+$
$L(t_7) = c-$

Fig. 4.4. A fragment with a merge place and a rule with a Boolean function.

structure in the level-ruled Petri net. The rule $r_1 \in R(t_4)$ uses syntactic abstraction to simplify the connectivity between transitions on a, b, and c. A combination of these transitions must fire for r_1 to be level satisfied. Suppose the following transitions exist for the level-ruled Petri net in Fig. 4.4: t_5, t_6, and t_7 such that $L(t_5) = a+$, $L(t_6) = b+$, and $L(t_7) = c-$. Note that these transitions are not shown in the drawn portion of the net in the figure. The rule r_1 is level satisfied if a transition from each of the following sets fire: $\{t_5, t_7\}$ and $\{t_6, t_7\}$. The set of these sets is a *level required set.* The level required set for r_2 is the empty set because its Boolean function is always true regardless of the Boolean state of the level-ruled Petri net.

The function $\mathsf{lrs}(t)$ returns the level required set for a transition t. The level required set is a minimal maxterm cover containing only prime implicants for the conjunction of the Boolean functions in the rule set of transition t. A maxterm prime implicant cover for t_4 is $(a \vee \neg c) \wedge (b \vee \neg c)$. The level required set for t_4 is $\{\{t_5, t_7\}, \{t_6, t_7\}\}$. Suppose there exists another transition t_8 such that $L(t_8) = a+$. The $\mathsf{lrs}(t_4)$ is now $\{\{t_5, t_7, t_8\}, \{t_6, t_7\}\}$. Appendix B is a more detailed discussion on the level required set. It is not presented here because it is not new research. Prime implicants are computed using standard algorithms and are then mapped to corresponding transitions. The challenge is to combine the marking required set with the level required set to create a set of causal groups for t_4.

A causal group only contains fired transitions. The *fired marking set*, T_μ, represents the fired transitions that created the current marking. The *fired Boolean state set*, T_ν, represents the fired transitions that created the current Boolean state. The construction of these two sets is different than that used to build the initial zone. It is presented in Section 4.3, but it is not needed for this part of the presentation. A causal group for a transition $T_c \subseteq T_\mu \cup T_\nu$ is a sufficient set of fired transitions to marking and level satisfy the transition.

Definition 4.18 (Causal Group). *A causal group given the fired marking set T_μ and fired Boolean state set T_ν for a given transition $t \in T$ is the set of transitions $T_c \subseteq T_\mu \cup T_\nu$ that satisfies the following:*

1. *for all transitions $t_c \in T_c$ there exists a $T_n \in \mathsf{lrs}(t) \cup \mathsf{mrs}(t)$ such that $t_c \in T_n$;*
2. *for all required sets $T_n \in \mathsf{lrs}(t) \cup \mathsf{mrs}(t)$, $T_c \cap T_n \neq \emptyset$; and*
3. *T_c is irredundant, meaning that nothing can be removed and still satisfy properties 1 and 2.*

A causal group for a transition t must satisfy three conditions. First, every transition in the causal group must belong to some member of either the level or marking required set; these contribute to marking and level satisfying transition t. Second, every marking and level necessary set must be satisfied by some transition in the causal group. A causal group for t_4 in Fig. 4.4 is $\{t_1, t_3, t_7\}$ for $T_\mu = \{t_1, t_3\}$, $T_\nu = \{t_5, t_6, t_7\}$, $L(t_5) = a+$, $L(t_6) = b+$, and $L(t_7) = c-$. Another causal group for transition t_4 using the same T_μ and T_ν is $\{t_1, t_3, t_5, t_6\}$. Third, and finally, T_c is irredundant. It is not possible to remove any transition from T_c and still have the first and second properties hold. This makes T_c a minimal set.

Generating causal groups for a transition is a unate covering problem. The covering problem is readily formulated from the marking required and level required sets after removing nonfired transitions. The marking required set for transition t_4 with only transitions from the fired marking set T_μ in the example is $\{\{t_1\}, \{t_3\}\}$. The level required set on the same set with only transitions from T_μ or the fired Boolean state set T_ν is $\{\{t_5, t_7\}, \{t_6, t_7\}\}$. Suppose that each transition t_i is now a binary variable. The marking required set can now be interpreted by the Boolean function $(t_1) \wedge (t_3)$ because both t_1 and t_3 are required by the rule set of transition t_4. The level required set can now be interpreted by the Boolean function $(t_5 \vee t_7) \wedge (t_6 \vee t_7)$. The two function must be true to level satisfy the rule set for t_4. The marking and level satisfied functions are combined to create the following Boolean function:

$$(t_1) \wedge (t_3) \wedge (t_5 \vee t_7) \wedge (t_6 \vee t_7).$$

This is the classical form of a unate covering problem. Any irredundant solution is a causal group for transition t_4.

Only irredundant solutions to the unate covering problem are included in the causal group set. The solution set for the above unate covering problem is

$$\{\{t_1, t_3, t_5, t_6\}, \{t_1, t_3, t_5, t_7\}, \{t_1, t_3, t_7\}\}$$

Consider the $\{t_1, t_3, t_5, t_7\}$ causal group. The presence of t_5 is not important in this causal group because $\{t_1, t_3, t_7\}$ is a solution to the covering problem too; thus, the $\{t_1, t_3, t_5, t_7\}$ solution is redundant and not included as a causal group according to Definition 4.18. The reason for this is best understood through example.

If the set $\{t_1, t_3, t_5, t_7\}$ is considered a causal group, then the time separation between the transitions in it and t_4 must fall within the bounds set by the rules in $R(t_4)$. This information is used to construct the equivalence classes. Although this is a topic of Section 4.3, it is briefly discussed here. Any transition is assumed to be able to fire last in this causal group to cause t_4 to be marking and level satisfied. Consider the t_5 member of this group. Assuming that t_5 fires after the other transitions in the causal group to marking and level satisfy t_4 is inconsistent. Transition t_4 is level satisfied when t_7 fires. Firing transition t_5 in this group is of no use. The optimal causal group set for a transition is the set of solutions to the unate covering problem with all of the solutions that form supersets of other solutions removed. Only two causal groups need to be considered in this example: $\{t_1, t_3, t_5, t_6\}$ and $\{t_1, t_3, t_7\}$.

The function $\mathsf{required\_set}(t, T_\mu, T_\nu)$ creates the required set for the covering problem. It takes three parameters: a transition t, the fired marking transitions T_μ, and the fired Boolean state transitions T_ν. The return value is the required set for t. The keep function must be defined before explaining $\mathsf{required\_set}(t, T_\mu, T_\nu)$.

Definition 4.19 (Keep Function). *The keep function keeps in the member sets of $\mathcal{T}$ any transitions not in T' and is* $\mathsf{keep}(T')(\mathcal{T}) = (\bigcup_{T'' \in \mathcal{T}} \{T'' \cap T'\})$.

The return value from $\mathsf{required\_set}(t, T_\mu, T_\nu)$ is $\mathsf{keep}(T_\mu)(\mathsf{mrs}(t)) \cup \mathsf{keep}(T_\nu)(\mathsf{lrs}(t))$. This is the union of the marking required set containing only fired marking transitions with the level required set containing only fired Boolean state transitions.

The function to create a causal group set is $\mathsf{unate\_solver}(\mathcal{T}_\mathsf{n}, T')$. It takes two parameters: a required set $\mathcal{T}_\mathsf{n}$ and a set of transitions T'. The required set, $\mathcal{T}_\mathsf{n}$, is the return value from $\mathsf{required\_set}(t, T_\mu, T_\nu)$ for some transition $t \in T$. The transition set T' is a set of transitions that are of interest in any solution to the unate covering problem in $\mathcal{T}_\mathsf{n}$. The $\mathsf{unate\_solver}(\mathcal{T}_\mathsf{n}, T')$ function returns the solutions to the unate covering problem in $\mathcal{T}_\mathsf{n}$. The function first computes the solutions, $\mathcal{T}_c$, to the covering problem. $\mathcal{T}_c$ is a set of transition sets with each set representing a solution that forms a causal group. The function then removes from each causal group transitions not in T' with the function $\mathsf{keep}(T')(\mathcal{T}_c)$. Finally, the function removes all redundant causal groups and returns. A redundant causal group is any group that is a superset of another group. Each causal group in the set now contains transitions found only in T'.

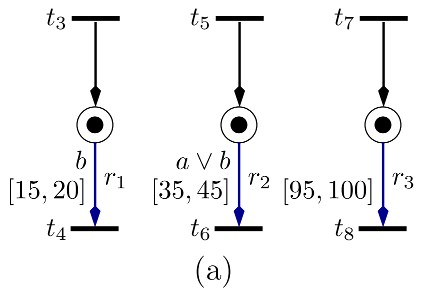
Any transition in a causal group of t can be the causal transition. The causal transition is the last transition to fire to reset the timer on a rule in the rule set of t. Each causal transition thus defines a maximum separation between itself and t. Causal groups and assignments are used to compute fireable transitions from a given timed state class. The term fireable is the timed state class counterpart to the timed state's enabled term. A fireable transition is an enabled transition in a timed state class. Computing the set of fireable transitions from a timed state class using causal group sets is the topic of Section 4.3.

4.3 Fireable Transitions

The set of fireable transitions in a timed state class relates to the set of enabled transitions in the timed state from Chapter 2. As a timed state can only fire enabled delay-transition pairs, only fireable transitions can fire from a timed state class to move the system to a new timed state class. Consider the level-ruled Petri net fragment in Fig. 4.5(a) and a corresponding timed state class $s = (\mu, \nu, z)$

such that the marking μ is the one shown in the figure, a and b are high in the Boolean state ν, and the zone z is the one shown in Fig. 4.5(b). The zone shows that transitions t_1 and t_2 on signals a and b fire between 60 and 75 time units after transition t_7. The zone also shows that transitions t_1 and t_2 fire within 15 time units of each other in either order. The rule sets for transitions t_4, t_6, and t_8 are marking and level satisfied in this timed state class. Which of the three transitions, however, can actually fire from this timed state class? This is an important question because it affects the correctness of the state space. If in all allowed firing sequences of the system, transitions t_4 and t_8 are the only transitions to fire from this marking and Boolean state, then these must be the only two transitions that can fire from this timed state class for the resulting finite state space representation to be correct. If transition t_6 is allowed to incorrectly fire from this timed state class, then the timing analysis introduces a behavior that does not exist in the model and the finite representation is less exact.

The earliest and latest firing times in the rules imply time separation bounds between transitions in the zone and the transitions that have marking and level satisfied rule sets. Transitions t_4, t_6, and t_8 have marking and level satisfied rule sets in Fig. 4.5(a). Transition t_4 has a single causal group $\{t_2, t_3\}$; transition t_6 has two causal groups $\{\{t_1, t_5\}, \{t_2, t_5\}\}$; and transition t_8 has a single causal group $\{t_4\}$. The rule r_1 for transition t_4 requires that t_4 be separated from t_2 and t_3



	t_7	t_1	t_2
t_7	0	-60	-60
t_1	75	0	15
t_2	75	15	0

$L(t_1) = a+$
$L(t_2) = b+$

(b)

Fig. 4.5. A level-ruled Petri net fragment with four potential fireable transitions with its zone. (a) The level-ruled Petri net fragment. (b) A zone with separations between t_7, t_1, and t_2.

by at least its earliest firing time of 15; and it requires that t_4 fire no more than 20 time units after either t_2 or t_3. The rules r_2 and r_3 create similar separations for t_6 and t_8 and their causal groups. A causality assignment is the selection of a causal group and causal transition within the group for each marking and level satisfied transition. The causality assignment for t_4 is t_2 in its only causal group; the assignment for t_6 is t_2 with its $\{t_2, t_5\}$ group; and the assignment for t_8 is t_7 in its only causal group. The *causal assigned zone* is the zone from the timed state class with all of the marking and level satisfied transitions added to it with separation bounds for their assignments. Fig. 4.6(a) is a causal assigned zone for transitions t_4, t_6, and t_8. The zone only defines separations between transitions in the zone and the causal assignment. The zone shows that transition t_4 fires between 15 and 20 time units after transition t_2; transition t_6 fires between 35 and 45 time units after t_2; and transition t_8 fires between 95 and 100 time units after t_7. Note that the zone also sets the separation between t_2 and t_1 to be 0. This implies that t_2 always fires before t_1 in this zone.

The set of fireable transitions is derived from a set of causal assigned zones because each transition with a marking and level satisfied rule set may have several causal assignments. The number of causal assignments to consider depends on the size and number of causal groups in the set of transitions with marking and level satisfied rule sets.

Definition 4.20 (Fireable). *A transition $t \in T$ is fireable in a given timed state class if it is first, in the set $T' \subseteq T$ of transitions with rule sets that are marking and level satisfied by the state; and second, using the zone in the state, there exists a causal assigned zone for transitions in T' such that its canonical form is consistent and allows t to fire before the other transitions in T'.*

The time satisfied portion of the enabled property in the timed state is replaced by the fire first property in the timed state class in this definition. The set of fireable transitions is the set of transitions that can fire concurrently from the timed state class considering all causal assigned zones. Consider again the causal assigned zone in Fig. 4.6(a). Its canonical form is shown in Fig. 4.6(b). The zone is consistent by Definition 4.5 and the set of concurrent transitions in the set of marking and level satisfied transitions $\{t_4, t_6, t_8\}$ is $\{t_4, t_8\}$ by Definition 4.14 as the zone shows that t_6 fires at least 15 time units after t_4 on this causal assignment. Although this causal assigned zone does not allow t_6 to fire concurrently, another one does exist on a different causal assignment to make it fireable too. The final fireable transition set for Fig. 4.5(a) given the zone in Fig. 4.5(b) is $\{t_4, t_6, t_8\}$. The goal of this section is to present an algorithm to compute the set of fireable events such that each transition in the set satisfies Definition 4.20.

4.3.1 Causal Assigned Zone

A causal assigned zone contains time separations between the newly added transition and its causal assignment. It also contains relations in the zone to make the causal assignment valid with respect to other possible causal assignments to the transition. The earliest and latest firing times must be defined in terms of a transition and its causal assignment to create the causal assigned zone. Recall that the *earliest firing time* $\mathsf{Eft} : R \rightarrow \mathbb{Q}^+$ is a function mapping rules to nonnegative rational numbers. For a rule $r = (p, t)$, $\mathsf{Eft}(r)$ is a minimum separation that must exist between the firing of t and the satisfaction of two conditions for r. The two conditions are marking and level satisfied. These conditions come about from firing transitions in the causal assignment for t. Transitions are connected to a rule

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	∞	∞	-95
t_1	75	0	15	∞	∞	∞
t_2	75	0	0	-15	-35	∞
t_4	∞	∞	20	0	∞	∞
t_6	∞	∞	45	∞	0	∞
t_8	100	∞	∞	∞	∞	0

(a)

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	75	-95	-95
t_1	75	0	15	0	-20	-20
t_2	75	0	0	-15	-35	-20
t_4	95	20	20	0	-15	0
t_6	120	45	45	30	0	25
t_8	100	40	40	25	5	0

(b)

Fig. 4.6. A causal assigned zone and its canonical form. (a) A causal assigned zone for transitions t_4, t_6, and t_8. (b) The canonical form of the causal assigned zone in (a).

through a place or through a Boolean function on the rule. The place connectivity comes from the flow relation in the level-ruled Petri net. The connection to the Boolean function stems from the dependence of the Boolean function on the signal associated with the transition. The function $\mathsf{lts}(r)$ returns the level transitions set that affects the truth value of the Boolean function associated with the rule $r \in R$. This is computed using positive and negative cofactors on the Boolean function for r. If a function is positive unate for a given signal, then it depends on that signal in its positive phase. The set $\mathsf{lts}(r)$ includes all rising transitions on a signal if a signal is positive unate in the function $\mathsf{Lsat}(r)$. Similar relations hold for negative unate and mixed unate. Complete details on the function $\mathsf{lts}(r)$ are found in Appendix B.

Definition 4.21 (Needs Set). *The needs set for a given rule* $(p, t) \in R$ *is* $\mathsf{needs}((p, t)) = \{t' \in T \mid t \in \mathsf{lts}((p, t)) \vee t \in \bullet p\}$*; it is extended to transitions as* $\mathsf{needs}(t) = \bigcup_{r \in R(t)} \mathsf{needs}(r)$.

This is the set of transitions that contribute to marking or level satisfying the rule.

Definition 4.22 (Earliest Firing Time). *The earliest time that transition* t_f *can fire after a transition* t_c *fires in its causal group is* $\mathsf{Eft}(t_c, t_f) = \max(\{q \in \mathbb{Q}^+ \mid \exists r \in R(t_f) : t_c \in \mathsf{needs}(r) \wedge \mathsf{Eft}(r) = q\})$.

The rules for transition t_f define the earliest time that t_f can fire after transition t_c. Transition t_c may be required by several rules in the rule set of transition t_f; thus, the earliest firing time of t_f after t_c is the maximum earliest firing time over the rules in the rule set of transition t_f that require transition t_c. Transition t_c is required by a rule r if it is in the set $\mathsf{needs}(r)$. Consider the level-ruled Petri net fragment in Fig. 4.4. The earliest firing time for the (t_3, t_4) transition pair is 2. For the level-ruled Petri net example in Fig. 4.5(a), the earliest firing time for the pairs (t_1, t_6), (t_2, t_6), and (t_5, t_6) is 35. Although Definition 4.22 uses the max function to resolve the case where a transition pair has multiple defined bounds, the timing analysis algorithm is only correct if the set has a single member (i.e., for a given t_c and t_f, $|\{q \in \mathbb{Q}^+ \mid \exists r \in R(t_f) : t_c \in \mathsf{needs}(r) \wedge \mathsf{Eft}(r) = q\}| = 1$). If a transition pair is involved in multiple rules, then those rules must share identical timing bounds for correctness.

The latest firing time for a transition pair is defined similarly to the earliest firing time. Recall that the *latest firing time* is defined as $\mathsf{Lft} : R \rightarrow \mathbb{Q}^+ \cup \{\infty\}$. The symbol ∞ represents an infinite latest firing time.

Definition 4.23 (Latest Firing Time). *The latest time that transition t_f can fire after a transition t_c fires in its causal group is $\mathsf{Lft}(t_c, t_f) = \max(\{q \in \mathbb{Q}^+ \mid \exists r \in R(t_f) : t_c \in \mathsf{needs}(r) \wedge \mathsf{Lft}(r) = q\})$.*

This definition is identical to the earliest firing time, excepting the use of $\mathsf{Lft}(r)$ instead of $\mathsf{Eft}(r)$. The value is again a maximum because transition t_c may be required by several rules in the rule set of t_f. Note that Definition 4.23 falls under the same restriction as Definition 4.22. If a transition pair is involved in multiple rules, then those rules must have identical timing bounds.

It is important to understand the relation between the earliest firing time on a rule and the earliest firing time on a transition pair. A rule requires transitions to fire to make it marking and level satisfied in a state. Consider the rule r_1 in Fig. 4.5(a). Suppose that the current state of the system, unlike the shown state, is such that rule r_1 is not marking satisfied, but the signal b is high in the Boolean state. The timer for rule r_1 is reset when transition t_3 fires in the level-ruled Petri net semantics because firing t_3 creates the state where rule r_1 becomes marking and level satisfied. The earliest firing time of transition t_4 after the firing of transition t_3 is 15 because firing t_3 resets the clock for r_1 and time must advance by at least 15 time units before the rule is time satisfied. If instead the timed state is such that firing transition t_2 on the signal b resets the clock for r_1 in the clock assignment function, then t_f cannot fire before 15 time units after t_2. The rule sets a lower bound for transitions required by the rule since each can be the last transition to fire to reset the clock on the rule in the timed state. A causal assigned zone sets the firing times of newly added transitions to not violate the earliest firing times for it and transitions in its causal group. It also sets the firing time of the newly added transition to fall within the latest firing time of its causal transition.

A causal assigned zone sets firing times to make the casual transition the last transition to fire with respect to transitions in the causal group that belong to rules that rely on the casual transition. Return to the two causal groups for transition t_4 in the level-ruled Petri net fragment in Fig. 4.4 assuming all transitions have fired except t_2. The causal groups are $\{t_1, t_3, t_5, t_6\}$ and $\{t_1, t_3, t_7\}$. If the causal group and assignment are $\{t_1, t_3, t_7\}$ and t_7, then a causal assigned zone sets t_7 to fire after transition t_1; thus, t_7 is the last transition to fire to reset the clock on the rule r_1. The latest time that t_4 can fire after t_7 is 5 time units. Making t_7 causal implies that it must be ordered with respect to transitions in rules that rely on it to become marking or level satisfied. The rule r_2 does not rely on t_7 to become marking or level satisfied; it only relies on t_3. In this sense, r_2 and thus, t_3, are independent of t_7 and need no ordering in this causal assignment.

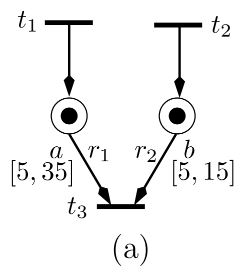
A causal transition is tied to specific rules. Consider the level-ruled Petri net in Fig. 4.7(a). The zone in Fig. 4.7(b) orders transition t_2 to always fire after t_1 and t_4. This does not, however, prevent t_1 or t_4 from being causal transitions because causality only implies the transition is the last transition to fire to reset the rule on a clock. The function $\mathsf{order\_set}(t_f, t_c, T_c)$ returns a set of transitions from the causal group T_c that belong to rules that rely on t_c to become either marking or level satisfied.

Definition 4.24 (Order Set). *The order set given a transition to fire $t_f \in T$, a causal group $T_c \subseteq T$, and a causal transition $t_c \in T_c$ is $\mathsf{order\_set}(t_f, t_c, T_c) = \{t \in T_c \mid \exists r \in R(t_f) : t \in \mathsf{TS}(r) \wedge t_c \in \mathsf{TS}(r)\}$; where r is the pair (p, t) and $\mathsf{TS}(r) = \mathsf{lts}(r) \cup \bullet p$.*

Note that the expression does not matter in computing the order set because it is considering only transitions in a given causal group. It only affects orders within this single conjunctive group. Orders between disjunctive groups are handled separately.

The causal group for Fig. 4.7(a) including only transitions in the zone shown in Fig. 4.7(b) is $T_c = \{t_1, t_4, t_2\}$. If t_4 is chosen to be the causal transition in this group, then the zone must be changed to fire t_4 after all transitions in $\mathsf{order\_set}(t_3, t_4, T_c)$, which is equal to $\{t_1, t_4\}$. This is the set of transitions from the causal group in rules that rely on t_4 to be marking or level satisfied. Transition t_1 must not be allowed to fire after t_4 in this example. This is done by setting the min-max entry on (t_1, t_4) to 0. Transition t_1 must now fire before or at the same time as t_4 ensuring that the firing of t_4 resets the clock on rule r_1. If t_2 is also forced to fire before t_4 in this zone, then it is no longer consistent in its canonical form, and t_4 is prevented from being causal. This is incorrect as t_4 can certainly define the separation between r_1 and t_3. If the causal transition, however, is t_2, then no orders in the zone need to be changed. Transition t_2 is always the transition to reset the clock on r_2 since a transition for b is not in the zone.

A causal assigned zone sets firing times to make transitions in the causal group actually be causal relative to the other causal groups. Consider again the causal group $\{t_1, t_3, t_7\}$ on the t_7 causal assignment to t_4 in the Fig. 4.4(a). The other possible causal group in this example is $\{t_1, t_3, t_5, t_6\}$. The causal zone must fire transitions in other causal groups such that t_7 is causal in the $\{t_1, t_3, t_7\}$ grouping. The transitions t_1 and t_3 are not important to making this group with t_7 causal because they are common to all causal groups. Transitions t_5, t_6, and t_7 are important because they are not common to every causal group. Firing t_7 before transitions t_5 or t_6 makes $\{t_1, t_3, t_7\}$ causal on t_7. Now consider the required set for t_4. This set is $\{\{t_1\}, \{t_3\}, \{t_5, t_7\}, \{t_6, t_7\}\}$. Although the members from



	t_1	t_4	t_2
t_1	0	5	-10
t_4	5	0	-5
t_2	15	20	0

$L(t_4) = a+$

(b)

Fig. 4.7. A level-ruled Petri net fragment with three potential fireable transitions with its zone. (a) The level-ruled Petri net fragment. (b) A zone with separations between t_1, t_4, and t_2.

the marking required set are singletons in this examples, this is not always the case. If the initial marking contains a merge place, then the fired marking set includes all transitions that can mark the merge place when fired. Recall that the set $\{t_5, t_7\}$ corresponds to the sum term $(t_5 \vee t_7)$ in the unate covering problem. Transition t_7 must fire before t_5 if it is the transition to satisfy this sum term. This corresponds exactly to one of the two orders on t_5, t_6, and t_7 necessary to make $\{t_1, t_3, t_7\}$ the causal group on transition t_7. A causal assignment can require several causal zones depending on the causal transition. There are two possible ways to make the $\{t_1, t_3, t_7\}$ group on t_7 valid in the example, but these orders are only important if t_7 is selected as the causal assignment. Consider again the required set $\{\{t_1\}, \{t_3\}, \{t_5, t_7\}, \{t_6, t_7\}\}$ for t_4. There are two causal assigned zones that must be created—a zone for each member of the required set where transition t_7 appears. The firing orders in each zone are set such that the causal transition t_7 fires before the other members in the sum term. Consider now the causal assignment of $\{t_1, t_3, t_7\}$ on transition t_1. This assignment creates one causal assigned zone because t_1 only appears in one member of the required set as a singleton.

The algorithm to create a causal assigned zone is shown in Fig. 4.8. There are five inputs to it: fired transition t_f, causal transition t_c, causal group T_c, required set $\mathcal{T}_\mathsf{n}$, and zone z. Transition t_f is the transition to fire. The causal transition for t_f is transition t_c, and its required set is $\mathcal{T}_\mathsf{n}$. Zone z is the zone from which the causal assigned zone is created. The return value is a set of causal assigned zones that includes the new transition t_f with appropriate orders and separations given t_c, T_c, and $\mathcal{T}_\mathsf{n}$.

There are four distinct operations performed by the algorithm in Fig. 4.8 on the zone after adding transition t_f to it. The operations correspond to the four requirements on the causal assigned zone: the fired transition must not violate the earliest firing time on members in T_c; it must satisfy the latest firing time on t_c, t_c must be the last transition to fire with respect to transitions in the causal group T_c that belong to rules that rely on the casual transition; and T_c must fire before transitions in other sum terms in $\mathcal{T}_\mathsf{n}$ that include T_c. The four operations

```
Algorithm: causal_assigned_zones($t_f, t_c, T_c, \mathcal{T}_\mathsf{n}, z$)
1: create zone $z'$ by adding $t_f$ to $z$
2: /* set earliest firing times for $t_f$ */
3: for all transitions $t$ in causal group $T_c$ do
4:     set min-max entry for $(t, t_f)$ in $z'$ to negative Eft$(t, t_f)$
5: /* set latest firing time of $t_f$ */
6: set max-min entry for $(t_f, t_c)$ in $z'$ to Lft$(t, t_f)$
7: /* make $t_c$ fire last to mark or level satisfy rules that rely on it */
8: for all transitions $t$ in order_set$(t_f, t_c, T_c)$ do
9:     set min-min entry for $(t, t_c)$ in $z'$ to 0 if positive in $z'$
10: /* make $T_c$ the causal group */
11: create empty zone set $Z$
12: for all transition sets $T'$ in required set $\mathcal{T}_\mathsf{n}$ where $t_c$ is in $T'$ do
13:     create zone $z''$ and set it equal to $z'$
14:     for all transitions $t$ in $T'$ do
15:         set min-min entry for $(t_c, t)$ in $z''$ to 0 if positive in $z''$
16:     add $z''$ to $Z$
17: return $Z$
```

Fig. 4.8. An algorithm to create causal assigned zones.

are demonstrated on the creation of a causal assigned zone for t_4 in the example in Fig. 4.4(a). The algorithm is called with $t_c = t_5$, $t_f = t_4$, $T_c = \{t_5, t_6\}$, $\mathcal{T}_\mathsf{n} = \{\{t_5, t_7\}, \{t_6, t_7\}\}$, and z is the one shown in Fig. 4.9(a). The causal group and required set for t_4 only contain transitions in the zone since these are the only transitions that can be part of a causal assignment.

Line 4 sets the earliest firing times for transitions in T_c and the fired transition t_f. Transition t_f fires at least 3 time units after transitions t_5 and t_6. If transition t_3 is in the zone too, then line 4 makes t_4 fire at least 2 time units after t_3. Line 6 sets the latest firing time. Transition t_4 cannot fire later than 5 time units after t_5. This completes the earliest and latest firing time operation on the zone.

Line 9 orders the causal transition to fire after other members of the order set order_set(t_f, t_c, T_c). Note that a once a causal assignment is made, it becomes a causal assignment for all rules that rely on it. The order set for this example is $\{t_5, t_6\}$. The min-min entry for (t_5, t_5) in the zone is already at 0. The min-min entry in the zone for (t_6, t_5) is 9; thus, t_6 can fire 9 time units after t_5 in this zone. This entry is set to 0. Transition t_6 can now only fire before or at the same time as t_5 making t_5 causal in this group. This completes the operation to make transition t_5 the reset transition for rules that rely on it. The zone after the first two operations is shown in Fig. 4.9(b). It contains the earliest and latest firing times and the new order on t_6 and t_5. The bounds for t_4 and the other members of the zone are infinite.

Line 15 orders the causal transition to fire before transitions in the other sum terms in the required set that include the causal transition. The required set contains a single sum term that includes t_5; the term is $\{t_5, t_7\}$. The min-min entry for (t_5, t_7) is 11 in the zone; thus, t_5 can fire 11 time units after t_7. This entry is set to 0. Transition t_5 must now fire before or at the same time as t_7. This is the only causal assigned zone for $\{t_5, t_6\}$ on t_5. The zone is shown in Fig. 4.9(c) with the new ordering on t_5 and t_7. If the causal assignment is $\{t_7\}$ on t_7, however, then two causal assigned zones exist—one zone for each member of $\mathcal{T}_\mathsf{n}$ containing t_7. These are shown in Fig. 4.10. The zone in Fig. 4.10(a) is where t_7 fires before t_5; and the zone in Fig. 4.10(b) is where t_7 fires before t_6.

4.3.2 Algorithm

The presentation thus far has defined the three necessary components to compute a set of fireable transitions. The first component is the timed state class. The second component is the causal group set. The third, and final, component is the

	t_5	t_6	t_7
t_5	0	15	11
t_6	9	0	7
t_7	2	5	0

(a)

	t_5	t_6	t_7	t_4
t_5	0	15	11	-3
t_6	0	0	7	-3
t_7	2	5	0	∞
t_4	5	∞	∞	0

(b)

	t_5	t_6	t_7	t_4
t_5	0	15	0	-3
t_6	0	0	7	-3
t_7	2	5	0	∞
t_4	5	∞	∞	0

(c)

Fig. 4.9. A series of three zones that show creation of a causal assigned zone. (a) The initial zone containing only transitions t_5, t_6, and t_7. (b) The zone with transition t_4 added and orders to make transition t_5 causal. (c) The final zone with transition t_5 ordered to be causal, earliest firing times for transitions t_5 and t_6, and a latest firing time for transition t_7.

	t_5	t_6	t_7	t_4
t_5	0	15	11	∞
t_6	9	0	7	∞
t_7	0	5	0	-3
t_4	∞	∞	5	0

(a)

	t_5	t_6	t_7	t_4
t_5	0	15	11	∞
t_6	9	0	7	∞
t_7	2	0	0	-3
t_4	∞	∞	5	0

(b)

Fig. 4.10. The two causal zones for the t_7 causal assignment. (a) Transition t_7 fires before t_5. (b) Transition t_7 fires before t_6.

causal assigned zone. The stage is set to present the algorithm to compute fireable transitions in a timed state class.

The set of fireable transitions is equivalent to the set of enabled transitions for a timed state. The algorithm to compute this set is shown in Fig. 4.11. This is a recursive algorithm that generates all possible causal assigned zones for a set of marking and level satisfied transitions. The set of marking and level satisfied fire first transitions are computed in each causal assigned zone and added to a set of fireable transitions. The algorithm takes five parameters. The first parameter,

```
Algorithm: fireable($T_s, T_{mls}, T_f, z$, FT)
 1: /* FT is the three-tuple ($T_\mu, T_\nu, T_z$) */
 2: if transition set $T_s$ is empty then
 3:     if zone $\check{z}$ is consistent then
 4:         add to $T_f$ fire first transitions from $T_{mls}$ in $\check{z}$
 5:     return $T_f$
 6: /* generate all causal assigned zones for this $t_f$ */
 7: remove a transition $t_f$ from $T_s$
 8: $\mathcal{T}_n$ = required_set($t_f, T_\mu, T_\nu$)
 9: create $\mathcal{T}_n'$ to be $\mathcal{T}_n$ with transitions not in $T_z$ removed
10: for all transition sets $T_c$ in unate_solver($\mathcal{T}_n, T_z$) do
11:     for all transitions $t_c$ in $T_c$ do
12:         for all zones $z_c$ in causal_assigned_zones($t_f, t_c, T_c, \mathcal{T}_n', z$) do
13:             $T_f$ = fireable($T_s, T_{mls}, T_f, z_c$, FT)
14:             if $T_f = T_{mls}$ then
15:                 return $T_f$
16: return $T_f$
```

Fig. 4.11. An algorithm that computes fireable transitions.

T_s, is the set of transitions that have yet to be causal assigned and added to the zone. The second parameter, T_{mls}, is the complete set of marking and level satisfied transitions. This set is necessary in computing fire first transitions in the causal assigned zone. The third parameter, T_f, is the set of fireable transitions as computed thus far by the algorithm. The fourth parameter, z, is the starting zone containing various causal transitions for members of the marking and level satisfied set T_{mls}. The fifth and final parameter, FT, is the *fired three-tuple.*

Definition 4.25 (Fired Three-tuple). *The fired three-tuple is* $\mathsf{FT} = (T_\mu, T_\nu, T_z)$ *where* T_μ*, is the set of transitions that fired to create the current marking,* T_ν *is the set of transitions that fired to create the current Boolean state, and* T_z *is the set of transitions in the current zone.*

The fired three-tuple is used to create the required set and remove from causal groups those transitions that are not in the zone.

The algorithm is explained using the level-ruled Petri net fragment in Fig. 4.5(a). The current timed state class for the level-ruled Petri net in Fig. 4.5(a) is the marking shown, the Boolean state is such that both a and b are high, and the zone is shown in Fig. 4.5(b). The zone shows that both t_1 and t_2 for $a+$ and $b+$ fired between 60 and 75 time units after t_7. It also shows that transitions t_1 and t_2 fire within 15 time units of each other. The set of marking and level satisfied transitions in the timed state class is $T_{mls} = \{t_4, t_6, t_8\}$. Function $\mathsf{fireable}$ is called with the set of unassigned transitions, T_s, set to T_{mls}. The fireable set, T_f is initially empty. The zone is the zone shown in Fig. 4.5(b). The fired three tuple is such that $T_\mu = \{t_3, t_5, t_7\}$, the $T_\nu = \{t_1, t_2\}$, and $T_z = \{t_1, t_2, t_7\}$.

The first part of the algorithm checks to see if all marking and level satisfied transitions have been causal assigned. If they have, then it computes the set of those transitions that can fire first from the causal assigned zone. The T_s set is not empty on the first call because not all transitions have been causal assigned. Control moves to line 7. Line 7 removes a transition from T_s. Line 8 computes the required set for the transitions. Transition t_4 is selected, and its required set $\mathcal{T}_\mathsf{n}$ is

$\{\{t_2\}, \{t_3\}\}$. Line 9 creates $\mathcal{T}_{\mathsf{n}}'$. This is a copy of $\mathcal{T}_{\mathsf{n}}$ but transitions not in T_z are removed from each member; $\mathcal{T}_{\mathsf{n}}'$ is $\{t_2\}$ in this example. The loop on line 10 iterates through all causal groups for t_4. There is a single causal group; it is $\{t_2\}$. Recall that $\mathsf{unate\_solver}(\mathcal{T}_{\mathsf{n}}, T_z)$ removes from the causal groups any transitions not in T_z. The loop on line 11 iterates on every causal transition in a causal group. Line 12 loops through each possible causal assigned zone given the causal assignment and required set $\mathcal{T}_{\mathsf{n}}'$. The causal assigned zone for t_4 is shown in Fig. 4.12(a). Line 13 makes the recursive call. This time T_s is $\{t_6, t_8\}$ and the zone is the causal assigned zone for t_4 in Fig. 4.12(a).

The set of transitions yet to be causal assigned, T_s, is still not empty so control moves to Line 7. Line 7 removes t_6. Line 8 computes its required set $\mathcal{T}_{\mathsf{n}}$; it has two members $\{t_5\}$ and $\{t_1, t_2\}$. The reduced version $\mathcal{T}_{\mathsf{n}}'$ is $\{\{t_1, t_2\}\}$. Line 10 iterates on the two possible causal groups $\{t_1\}$ and $\{t_2\}$. Line 11 picks the only transition in the second causal group $\{t_2\}$, and line 12 creates the causal assigned zone shown in Fig. 4.12(b). Notice that t_2 is ordered to fire before t_1 to make it causal (i.e., min-min first entry for (t_2, t_1) is 0). Line 13 makes the recursive call, only this time T_s is $\{t_8\}$, and the zone is the one shown in Fig. 4.12(b).

The set of transitions yet to be causal assigned, T_s, is still not empty so control moves directly to line 7. The last transition, t_8 is removed from T_s. Its required set from line 8 has a single member $\{t_7\}$. This results in a single causal assignment for t_8. The causal assigned zone for t_8 is shown in Fig. 4.12(c). This is sent to the recursive call on line 13 with the now empty set T_s.

The set of transitions yet to be causal assigned is empty and control moves to line 3 on this call. The canonical form of the zone in Fig. 4.12(c) is shown in Fig. 4.12(d). The zone is consistent because its diagonal is zero. Line 4 adds to the set T_f all fire first transitions in the zone given the marking and level satisfied transitions T_{mls}. The set of fire first transitions from $\{t_4, t_6, t_8\}$ is $\{t_4, t_8\}$ from Definition 4.14. Although it is not explicitly checked, the causal assigned zone must be consistent with respect to the fire first transitions and existing transitions in the zone according to Definition 4.5. Transition t_4 is not included in this set because

	t_7	t_1	t_2	t_4
t_7	0	-60	-60	∞
t_1	75	0	15	∞
t_2	75	15	0	-15
t_4	∞	∞	20	0

(a)

	t_7	t_1	t_2	t_4	t_6
t_7	0	-60	-60	∞	∞
t_1	75	0	15	∞	∞
t_2	75	0	0	-15	-35
t_4	∞	∞	20	0	∞
t_6	∞	∞	45	∞	0

(b)

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	∞	∞	-95
t_1	75	0	15	∞	∞	∞
t_2	75	0	0	-15	-35	∞
t_4	∞	∞	20	0	∞	∞
t_6	∞	∞	45	∞	0	∞
t_8	100	∞	∞	∞	∞	0

(c)

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	-75	-95	-95
t_1	75	0	15	0	-20	-20
t_2	75	0	0	-15	-35	-20
t_4	95	20	20	0	-15	0
t_6	120	45	45	30	0	25
t_8	100	40	40	25	5	0

(d)

Fig. 4.12. The evolution of a zone in the fireable algorithm as transitions are added. (a) The zone after t_4 is added with t_2 as its causal assignment. (b) The zone after t_6 is added with t_2 as its causal assignment. (c) The zone after t_8 is added with t_7 as its causal assignment. (d) The causal assigned zone in its canonical form.

the min-max entry for (t_4, t_6) in the zone is -15. This means that transition t_6 must always fire at least 15 time units after t_4. The new set of fireable transitions T_f is returned from this call.

Control returns to line 13 on the previous stack frame after the new fireable set is computed. There are no more causal assigned zones or causal assignments to consider for t_8. The new fireable set is again returned and control moves to line 13 on the previous stack frame. There is another causal assignment to consider to t_6. This is $\{t_1\}$. The zone created for this causal assignment is shown in Fig. 4.13(a). Compare this zone to the one shown in Fig. 4.12(b). Transition t_6 is now causal on t_1, and t_1 is ordered to fire before t_2 (i.e., the min-max entry for (t_1, t_2) in the zone is 0).

The recursive call is made to add transition t_8 into the zone on its only causal assignment. The recursive call is then made with the now empty set T_s and control moves to line 3. The canonical form of this new causal assigned zone is shown in

Fig. 4.13(b). It is consistent. The fireable transitions in this zone are t_4, t_6, and t_8. These are added to the fireable set, and it is returned.

The final set of fireable transitions is $\{t_4, t_6, t_8\}$. This is why all causal assignments must be explored for all transitions. A transition may be excluded by one causal assignment, but included by another. It is necessary to be able to compute all possible fireable transitions from a given timed state class; thus, all assignments must be explored. Although this number seems to be large, in practice, there are often not many causal assignments to explore because the number of transitions in the zone is small. In addition, it is often the case that all transitions are added on the first few causal assignments, or the assignment shows that certain transitions can never fire first. Now that a set of fireable transitions is known, however, it is necessary to compute successor timed state classes that result from firing one of these transitions. This is the topic of the next section.

4.4 Successor Transitions

Creating successor timed state classes by firing a transition from the fireable set is similar to computing the fireable set. The necessary components are the causal group and the causal assigned zone. Any transition may have several causal groups depending on its connectivity, implied or direct, in the level-ruled Petri net. Any causal group may produce several causal assigned zones depending on its required set. Each of these zones is a successor to the timed state class . This section presents

	t_7	t_1	t_2	t_4	t_6
t_7	0	-60	-60	∞	∞
t_1	75	0	0	∞	-35
t_2	75	15	0	-15	∞
t_4	∞	∞	20	0	∞
t_6	∞	45	∞	∞	0

(a)

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	-75	-95	-95
t_1	75	0	0	-15	-35	-20
t_2	75	15	0	-15	-20	-20
t_4	95	35	20	0	0	0
t_6	120	45	45	30	0	25
t_8	100	40	40	25	5	0

(b)

Fig. 4.13. A causal assigned zone and its canonical form. (a) The zone using transition t_2 as the causal assignment for transition t_6. (b) The canonical form of the causal assigned zone.

the algorithm to compute successor timed state classes by firing transitions from the fireable set.

The successor timed state classes are the result of firing a transition on every causal assignment given a starting timed state class.

Definition 4.26 (Successor). *Firing a marking and level satisfied transition $t \in T$ from a timed state class (μ, ν, z) with a given fired three-tuple FT results in a set of timed state classes S; each member $(\mu', \nu', z') \in S$ is such that μ' and ν' are the updated marking and Boolean state after firing t, and z' is a canonical, valid, and consistent causal assigned zone for t; the set S contains a member for each causal assigned zone for a given fired three-tuple FT.*

The successor set is readily computed using previously defined algorithms.

The algorithm to compute successor timed state classes from a given timed state class is shown in Fig. 4.14. The algorithm takes three parameters. The first parameter, t_f, is the transition to fire. The second parameter, s is the current timed state class. This is the three-tuple (μ, ν, z). The third, and final, parameter, FT,

```
Algorithm: successor($t_f, s, \mathsf{FT}$)
 1: /* $s$ and FT are the three-tuples $(\mu, \nu, z)$ and $(T_\mu, T_\nu, T_z)$ */
 2: create $\mu'$ and $\nu'$ by firing $t_f$ from $\mu$ and $\nu$
 3: if untimed_failure($\mu, \nu, t_f, \mu', \nu'$) then
 4:    report failure and exit
 5: create empty timed state class set $S$
 6: /* create all causal assigned zones for $t_f$ */
 7: $\mathcal{T}_\mathsf{n}$ = required_set($t_f, T_\mu, T_\nu$)
 8: set $\mathcal{T}_\mathsf{n}'$ to $\mathcal{T}_\mathsf{n}$ with transitions not in $T_z$ removed
 9: for all transition sets $T_c$ in unate_solver($\mathcal{T}_\mathsf{n}, T_z$) do
10:    for all transitions $t_c$ in $T_c$ do
11:       for all zones $z_c$ in causal_assigned_zones($t_f, t_c, T_c, \mathcal{T}_\mathsf{n}', z$) do
12:          if $\check{z}_c$ is consistent and valid then
13:             if timed_failure($t_f, (\mu, \nu, \check{z}_c), \mathsf{FT}$) then
14:                report timing failure and exit
15:             add timed state class $(\mu', \nu', \check{z}_c)$ to $S$
16: return $S$
```

Fig. 4.14. An algorithm that computes successor timed state classes.

is the fired three-tuple given as (T_μ, T_ν, T_z). The first two members represent the transitions that fired to create the current marking and Boolean state, respectively. The third member is the set of transitions currently in the zone.

The algorithm is explained using the level-ruled Petri net fragment in Fig. 4.5(a). The current timed state class for the level-ruled Petri net in Fig. 4.5(a) is the marking shown, the Boolean state is such that both a and b are high, and the zone is shown in Fig. 4.5(b). The zone shows that both t_1 and t_2 for $a+$ and $b+$ fired between 60 and 75 time units after t_7. It also shows that transitions t_1 and t_2 fire within 15 time units of each other. The set of fireable transitions in this timed state class is $T_{mls} = \{t_4, t_6, t_8\}$. The successor function is called with t_f set to t_6. The fired three tuple is such that $T_\mu = \{t_3, t_5, t_7\}$, the $T_\nu = \{t_1, t_2\}$, and $T_z = \{t_1, t_2, t_7\}$.

The algorithm is iterative. Line 2 creates the new marking and Boolean state that results from firing transition t_f by Definition 2.3 and Definition 2.23. Line 3 checks for any untimed failures that are caused by firing t_f. The failure function is formalized in Definition 3.22. It checks that the transition is safe, consistent state assigned, output semimodular, and constraint satisfied in the untimed sense. If the firing of t_f violates any of these properties, then it returns a failure. The failure is reported on line 4. Line 5 of the algorithm creates an empty set of timed state classes for the successors of this timed state class. Line 7 gets the required set for the transition t_f. Recall that transitions in member sets that are not in the T_μ and T_ν members of the fired three-tuple are removed from the required set. The required set for transition t_6 in this example is $\{t_1, t_2\}$. Line 9 iterates through the causal groups created by the required set. There are two causal groups for this example: $\{t_1\}$ and $\{t_2\}$. Line 10 iterates on all causal assignments. This example starts with the first causal assignment of $\{t_1\}$ on t_1. Line 11 iterates on the only causal assigned zone for this assignment. This is shown in Fig. 4.15(a). The canonical form of this causal assigned zone is shown in Fig. 4.15(b). Line 12 checks that the zone is consistent and valid. A zone is not consistent or valid if the causal assignment orders the transition to where it cannot fire, or it fires before other fired transitions. This zone is both consistent, because its diagonal is zero, and valid,

	t_7	t_1	t_2	t_6
t_7	0	-60	-60	∞
t_1	75	0	15	∞
t_2	75	0	0	-35
t_6	∞	∞	45	0

(a)

	t_7	t_1	t_2	t_6
t_7	0	-60	-60	-95
t_1	75	0	15	-20
t_2	75	0	0	-35
t_6	120	45	45	0

(b)

Fig. 4.15. A causal assigned zone for transition t_6 on t_2 and its canonical form. (a) The zone using transition t_2 as the causal assignment for transition t_6. (b) The canonical form of the causal assigned zone.

because transition t_6 fires after transitions already in the zone, by Definition 4.6. Line 13 checks for timing failures in constraint rules. This check is discussed in detail in Section 4.7. A failure occurs in a constraint rule for t_f if it is visible to the target module, and it is outside of its timing bounds at the firing of t_f, or if any marking and level satisfied constraint rule is beyond its upper timing bound. These properties are given in Definition 3.16 and Definition 3.17. If there is no timing failure, then line 15 combines the zone with the new marking and Boolean state to form a timed state class and adds it to S.

This algorithm implements a partial order in the timing information. In traditional timing analysis algorithms, the transitions in the zone need to be ordered to always fire before t_6. Transitions t_7, t_1, and t_2 are already ordered before t_6 in this example. This is shown in the t_6 column of the zone. It is all negative. Although it is ordered in this zone, a total order method modifies the t_6 column as necessary to change any positive entries to zero. This forces t_6 to always fire after transitions already in the zone. This order is not enforced by this algorithm. It allows members of the zone to fire after t_6 if possible (i.e., the t_6 column can have positive nonzero entries). This partial order in the timing information is the basis of the POSET method in [43, 57].

The next iteration in the algorithm creates the causal assigned zone shown in Fig. 4.16(a). Its canonical form is shown in Fig. 4.16(b). This zone too is combined with the new marking and Boolean state to form a timed state class and added to S making the final set returned from the successor algorithm after firing transition

	t_7	t_1	t_2	t_6
t_7	0	-60	-60	∞
t_1	75	0	0	-35
t_2	75	15	0	∞
t_6	∞	45	∞	0

(a)

	t_7	t_1	t_2	t_6
t_7	0	-60	-60	-95
t_1	75	0	0	-35
t_2	75	15	0	-20
t_6	120	45	45	0

(b)

Fig. 4.16. A causal assigned zone for transition t_6 on t_1 and its canonical form. (a) The zone using transition t_1 as the causal assignment for transition t_6. (b) The canonical form of the causal assigned zone.

t_6. Comparing the two zones in Fig. 4.15(b) and Fig. 4.16(b) is of interest. The ordering necessary to create the two causal assignment splits zones so that they do not form a subset or superset relation with each other.

It is possible to add a timed state class that leads to a timed state classes in the future with no successors. Recall from the previous section on computing fireable events that one of the two causality assignments for the marking and level satisfied set of $\{t_4, t_6, t_8\}$ in Fig. 4.7(a) only allowed transitions t_4 and t_8 to fire. The successor algorithm adds the two timed state classes in Fig. 4.15(b) and Fig. 4.16(b) after firing transition t_6. Consider the zone in Fig. 4.16(b) on the t_2 causality assignment for t_6. Suppose that after firing t_6 it is determined that transition t_1 is no longer needed in the zone; thus, it is deleted. The resulting zone is used to form the timed state class after t_6 fires. The two fireable transitions from this state are t_4 and t_8. Suppose that t_8 fires to generate one successor zone. After firing t_8, transition t_7 is no longer needed in the zone, so it is deleted. At this point, transition t_4 is the only marking and level satisfied transition. It is fired by the successor function to create the zone in Fig. 4.17. This zone, however, is not valid because the min-max

	t_2	t_6	t_8	t_4
t_2	0	-35	-20	-15
t_6	45	0	25	30
t_8	40	5	0	25
t_4	20	-15	0	0

Fig. 4.17. A causal assigned zone that is not valid.

entry for (t_4, t_6) is -15. This zone has ordered the newly added transition t_4 to fire before transition t_6. This violates the implied order in the zone, and the zone is discarded; thus, there are no successors from this timed state class if t_4 is the only marking and level satisfied transition.

The intuition behind this issue is that the set of fireable transitions is computed from all possible causal assignments to a group of marking and level satisfied transitions. Each causal assignment can generate a different set of fireable transitions. The fireable function, however, creates the set of fireable transitions as the union over the sets generated in each causal assignment. This implies that it is possible to fire a transition first from a causal assignment that does not actually allow that transition to fire first. This leads to states with no successors. This cannot be avoided without fundamentally changing the approach of the algorithm. It would need push onto the search stack each zone created in fireable on causal assignment. The search then only fires true fire first transitions according to each causal assignment across the set of marking and enabled transitions.

4.5 Finite Representation

The stage is set to build a finite representation of the infinite state space for a level-ruled Petri net. The timed state class is a suitable replacement for the timed state. Its zone can capture many different clock assignment functions in the timed state. The causal group allows both fireable and successor states to be computed from a given timed state class. All that is missing is a representation and algorithm to capture the reachable states and firing sequences of the system using the timed state class.

The first section presents the timed state class graph. The infinite state space is captured in a graph representation. The graph, however, is unique in that the edge relation only considers the marking and Boolean state while the node set consists of timed state classes. The concluding section presents an algorithm to build the timed state class graph. It uses, as mentioned previously, the fireable and successor algorithms.

4.5.1 Timed State Class Graph

The timed state class graph is a finite representation of an infinite set of timed states and firing sequences.

Definition 4.27 (Timed State Class Graph). *A timed state class graph is the tuple* $\mathrm{SG} = (S, E)$ *where* S *is a set of timed state classes and* $E \subseteq 2^P \times 2^W \times T \times 2^P \times 2^W$ *is the edge relation.*

The nodes of the graph are timed state classes. The edges of the graph are represented by a relation. The relation, however, only links the marking and Boolean state to other markings and Boolean states by fired transitions. The zone is excluded in the relation because it can be derived from the set of reachable timed state classes S in the system. For any given edge in E, there exists a corresponding timed state class pair in S that satisfies the transition—meaning the transition is fireable in the first member of the pair, and there is a causal assignment that leads it to the second member of the pair.

A timed state class is not new in the node set unless it has a marking or Boolean state that has yet to be seen, or its zone is not a subset of an existing zone for the same marking and Boolean state in the node set.

Definition 4.28 (New Timed State Class). *A timed state class* (μ, ν, z) *is new in a timed state class set* S *if for all states* $(\mu', \nu', z') \in S$, $\mu \neq \mu' \vee \nu \neq \nu'$*; and for all states* $(\mu', \nu', z') \in S$, $\mu = \mu' \wedge \nu = \nu' \wedge z \not\subseteq z'$*.*

The new timed state class definition does not allow the node set to be larger than it needs to be. In only keeps zones that cover clock assignment functions that have yet to be seen.

Definition 4.29 (Adding A State). *Adding a new timed state class* (μ, ν, z) *to a timed state class set* S *results in the new timed state class set* $S' = (S - \{(\mu', \nu', z') \in S \mid \mu = \mu' \wedge \nu = \nu' \wedge z \supset z'\}) \cup \{(\mu, \nu, z)\}$*.*

Adding a new timed state class can actually shrink the size of the finite representation. This is because any timed state classes with identical markings and Boolean states but containing subset zones are removed from the node set. The relation E for the edge set becomes more clear at this point. It never needs to be updated when the node set is reduced by the superset relation because it does not include the zone.

The update on the edge relation E is straight forward. The new edge is added to the relation.

Definition 4.30 (Adding An Edge). *Adding a new edge* $(\mu, \nu, t, \mu', \nu')$ *to an edge set* E *results in the new edge set* $E' = E \cup \{(\mu, \nu, t, \mu', \nu')\}$.

4.5.2 Algorithm

The algorithm cannot be presented without a brief discussion of the fired three-tuple. The fired three-tuple is given in Definition 4.25. The three members of the tuple represent sets of fired transitions. The first is the set of transitions that created the current marking. The second is the set of transitions that created the current Boolean state. The third, and final, is the set of transitions in the current zone. The three sets are important to computing the causal group set and causal assigned zones. The first two members reduce the size of the covering problem that needs to be solved. They remove from the required set any transitions that have not fired. A solution containing these transitions does not make sense since the transitions have not fired. The third member is used to remove transitions that are not in the zone. The separations between these transitions and those in the zone are unconstrained; they are not considered in computing fireable and successor transitions.

The fired marking transition set is updated with the firing of a new transition by looking at the fired transition and the previous fired marking set. It cannot be determined by the marking alone. This is due to merge places. The marking does not contain information on the transition that fired to add a merge place to the marking.

Definition 4.31 (Initial Fired Marking Transition Set). *The initial fired* *marking set of a level-ruled Petri net with the initial marking* μ_o *is* $T_\mu = \{t \in T \mid \exists p \in \mu_o : t \in \bullet p\}$.

Consider the level-ruled Petri net fragment in Fig. 4.4. If the initial marking is such that the rules r_1 and r_2 are marking satisfied, then it is not known if t_1 or t_2 fired to create the marking; thus, both of these transitions are included in the initial fired marking set.

Definition 4.32 (Update on Fired Marking Transitions). *Adding a fired transition* $t_f \in T$ *to the set of fired marking transitions* T_μ *given the new marking* μ *after* t_f *fires creates the new set* $T'_\mu = \{t \in T \mid t = t_f \vee (t \in T_\mu \wedge \exists p \in \mu : t \in \bullet p)\}$.

The new fired marking set only includes transitions connected to places in the new marking, and it does not include any transitions that share a place with t_f; thus, aside from the initial set, no more than one transition involved in a merge place is ever included.

The fired Boolean state set must deal with multiple transitions on the same signal. This affects the initial fired Boolean state set as well as the update operation.

Definition 4.33 (Initial Fired Boolean State Transition Set). *The initial fired Boolean state set of a level-ruled Petri net given its initial Boolean state* ν *is* $T_\nu = \{t \in T \mid (L(t) = w+ \wedge w \in \nu_o) \vee (L(t) = w- \wedge w \notin \nu_o)\}$.

The initial fired Boolean state transition set includes all transitions that contributed to creating the initial Boolean state. If multiple transitions exist for a given signal that move that signal to the same Boolean state that matches the state in ν_o, then all of those transitions are included in the initial set. The system cannot know which of those transitions fired to create the initial Boolean state ν_o.

Definition 4.34 (Update on Fired Boolean State Transitions). *Adding a fired transition* $t_f \in T$ *to the set of fired Boolean state transitions* T_ν *given the new Boolean state from firing* t_f *creates the new set* $T'_\nu = \{t \in T \mid ((L(t) = w+ \vee L(t) = w-) \wedge t = t_f) \vee (L(t_f) = w+ \wedge L(t) \neq w- \wedge t \in T_\nu) \vee (L(t_f) = w- \wedge L(t) \neq w+ \wedge t \in$

$T_\nu)\}$.

The update on the fired Boolean state set is similar to the update on the fired marking transition set. It removes from the set transitions on the same signal as t_f but in the opposite direction. There are three conditions that a transition can satisfy to be added to the updated set. The first case is for the actual fired transition. If the fired transition t_f is on a signal, then add it to the updated set. The second and third conditions relate to transitions already in the set. If the fired transition t_f is rising (falling), transition t is not a falling (rising) transition on the same signal, and t is in the old Boolean state fired set, then add it to the updated set.

The zone transition set is simpler to compute than the fired marking and Boolean state transitions sets.

Definition 4.35 (Zone Transitions). *The set of transitions in a given zone* $z = (\mathbf{t}, A)$ *is the set of transitions* $T_z = \{t \in T \mid \exists i : t_i = t\}$.

The initial set is computed using the same definition. A transition is in the set if it exists in the zone. Multiple instances are not an issue.

The algorithm to build the finite representation of the state space given a level-ruled Petri net is shown in Fig. 4.18. This is a depth first search algorithm. The

```
Algorithm: find($s$, SG, FT)
 1: /* Recall that $s$ is the three-tuple $(\mu, \nu, z)$ */
 2: create the set $T_{mls}$ of all $(\mu, \nu)$ satisfied transitions
 3: for all transitions $t_f$ in fireable($T_{mls}, T_{mls}, \emptyset, z$, FT) do
 4:    for all timed state classes $s'$ in successor($t_f, s$, FT) do
 5:       add edge $(s, t_f, s')$ to timed state class graph SG
 6:       $s'$ = prune($s'$, FT)
 7:       if $s'$ is new in SG then
 8:          update the fired tuple FT with $t_f$ and $z'$
 9:          add $s'$ to SG
10:          SG = find($s'$, SG, FT)
11: return SG
```

Fig. 4.18. An algorithm to find a timed state class graph.

algorithm takes three parameters. The first parameter, s, is the current timed state class of the level-ruled Petri net. The second parameter, SG, is the timed state class graph. The third, and final, parameter is the fired three-tuple. The return type for the algorithm is a timed state class graph that represents the reachable state space of the level-ruled Petri net. The first call to the algorithm is with the initial timed state class $s = (\mu_o, \nu_o, z_o)$, where μ_o and ν_o come directly from the level-ruled Petri net, and z_o is derived from μ_o and ν_o according to Definition 4.16. The initial timed state class graph SG contains only the initial timed state class in its node set. The initial fired three-tuple is derived from μ_o, ν_o, and z_o, according to the above definitions.

There are two operations in the algorithm that have not been defined thus far. The first operation is on line 2. The T_{mls} set is the set of marking and level satisfied transitions in the timed state class (μ, ν, z). This set is given as $\{t \in T \mid (\mu, \nu) \vdash R(t)\}$. The other undefined operation is on line 6. The function $\mathsf{prune}(s', \mathsf{FT})$ deletes from a zone in a timed state class any transitions that are no longer important in computing fireable and successor states. A transition is not important if it is no longer part of the fired marking or Boolean state sets. A transition is also not important if it is determined that it no longer can be causal to a transition that requires it to be marking or level satisfied. The number of transitions in the zone affects the performance of the algorithm. The algorithm generally has better performance in both its running time and the size of the final timed state class graph if the zones contain the fewest number of transitions as possible; thus, the function $\mathsf{prune}(s', \mathsf{FT})$ is important to the success of the algorithm.

4.6 Pruning

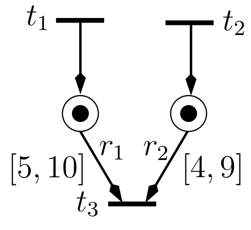
Pruning reduces the number of transitions in the zone to improve the timing analysis algorithm in Fig. 4.18. Zones containing fewer transitions reduce the running time of the algorithm because fewer relations need to be considered in the fireable and successor algorithms. Pruning reduces the size of the timed state class graph because the zones are larger and more apt to form superset relations.

The next three sections present the pruning method used in timing analysis to build the finite state space representation.

4.6.1 Type I Redundant Transitions

Pruning removes transitions that no longer contribute information in the zone useful to creating successor states. The intuition is best understood through example. Consider the level-ruled Petri net in Fig. 4.19(a). Suppose the zone in the current timed state class is the one shown in Fig. 4.19(b). Do both of the transitions need to remain in the zone? The question is addressed by the graph in Fig. 4.19(c). The graph shows the placement of transition t_3 in time. Transition t_1 serves as a reference point in the graph. Transition t_2 is redundant in this zone if the earliest and latest firing times of t_3 are completely determined by t_1.

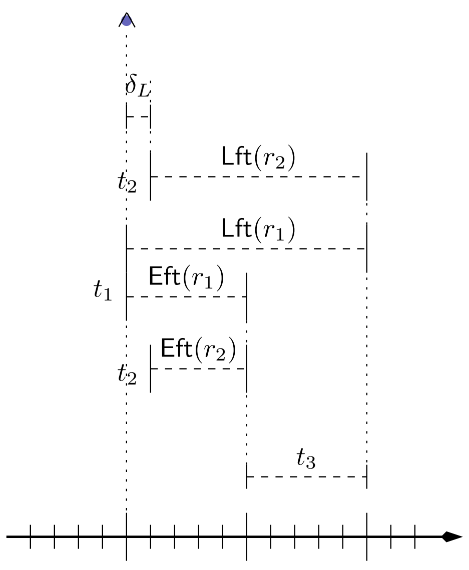
The maximum amount of time that can elapse from the firing of t_1 to t_2 is the min-max entry (t_2, t_1) in the zone. Transition t_2 can fire as much as 1 time unit after t_1. This is the δ_L separation in the graph. It represents firing transition t_1 as early as the zone allows and then firing transition t_2 as late as the zone allows;



(a)

	t_1	t_2
t_1	0	2
t_2	1	0

(b)



(c)

Fig. 4.19. A level-ruled Petri net, zone, and graph to illustrate pruning. (a) The level-ruled Petri net fragment. (b) A zone with separations between t_1 and t_2. (c) A graph showing the placement of t_3 in time depending on its causal assignment.

hence, the δ_L indicates that t_2 fires as late as possible relative to the reference point t_1. Transition t_1 completely determines the latest firing time of t_3 if $\delta_L + \mathsf{Lft}(r_2) \leq \mathsf{Lft}(r_1)$. Transition t_2 cannot fire any later than δ_L after t_1; thus, if the latest firing time for t_3 set by r_1 is after that set by r_2 at its latest firing time, then surely t_1 completely determines the latest firing time of t_3. This is shown in the graph by making t_2 fire earlier than where it is shown. This effectively moves it to the left in the graph. Notice that the latest firing time of t_3 is not changed as t_2 fires earlier in time relative to t_1.

The earliest firing time is checked in a similar fashion. Transition t_1 completely determines the earliest firing time of t_3 if $\delta_L + \mathsf{Eft}(r_2) \leq \mathsf{Eft}(r_1)$. Transition t_2 cannot fire any later than δ_L time units after t_1; thus, if the earliest firing time set by r_1 is after that set by r_2 with t_2 at its firing as late as possible, then surely t_1 completely determines the earliest firing time of t_3. This is shown in the graph by making t_2 fire earlier than where it is shown. This effectively moves it to the left in the graph. Notice that the earliest firing time of t_3 is not changed as t_2 fires earlier in time relative to t_1. Transition t_1 completely determines the earliest and latest firing time of t_3 in this example. This is correct because both the earliest and the latest firing times are set by a max relation over the rules.

The intuition in the graph is supported by the causal assigned zones. The remaining transition in the zone is sufficient to capture all the allowed behaviors in the successor states of the level-ruled Petri net. Consider the two causal assigned zones created from the t_1 and t_2 causal assignments in this example. Fig. 4.20(a) is the zone from the t_1 causal assignment and Fig. 4.20(b) is the zone from the t_2 causal assignment. Both zones are in their canonical forms. The zone in Fig. 4.20(a) is a superset of the zone in Fig. 4.20(b); thus, computing causality using only t_1 captures the earliest and latest possible firing time of t_3. The reduced zone with t_2 deleted is shown in Fig. 4.20(c). The reduced zone correctly sets the earliest and latest firing times of t_3.

The required set is part of the redundant calculation. The required set in the above example is very simple because it only has two size one members. Consider

	t_1	t_2	t_3
t_1	0	2	-5
t_2	1	0	-4
t_3	10	12	0

(a)

	t_1	t_2	t_3
t_1	0	2	-5
t_2	1	0	-4
t_3	10	9	0

(b)

	t_1	t_3
t_1	0	-5
t_3	10	0

(c)

Fig. 4.20. Two causal assigned and a reduced zone for Fig. 4.19. (a) The causal assigned zone for the t_1 assignment. (b) The causal assigned zone for the t_2 assignment. (c) The reduced causal assigned zone for t_3.

the required set for t_4 in Fig. 4.4: $\{\{t_1, t_2\}, \{t_3\}, \{t_5, t_7\}, \{t_6, t_7\}\}$. This is more complex in that several members contain more than one transition. How does this impact the redundant calculation? The answer is found in the causal assigned zone.

Definition 4.36 (Type I Redundant Transition). *A transition, $t_p \in T_z$, is redundant to a transition yet to fire, $t_f \in T$, in the zone z defined over $T_z \subseteq T$ if one of the following holds:*

1. *for all $T' \in \mathsf{required\_set}(t_f, T_z, T_z)$, $t_p \notin T'$;*
2. *for all $T' \in \mathsf{required\_set}(t_f, T_z, T_z)$ such that $t_p \in T'$, t_p is not a fire first transition in T' given z; or*
3. *there exists $T' \in \mathsf{required\_set}(t_f, T_z, T_z)$ such that $t_p \notin T'$ and for all transitions $t \in T'$, $\delta_L + \mathsf{Lft}(t_p, t_f) \leq \mathsf{Lft}(t, t_f) \wedge \delta_L + \mathsf{Eft}(t_p, t_f) \leq \mathsf{Eft}(t, t_f)$; where δ_L is the min-max entry for (t_p, t) in z.*

The first case simply checks that t_p is connected to t_f through a marking or Boolean function. The second case is a result of the causal assigned zone. If t_p is a member of a required set that contains more than one transition, then it must be able to fire before the transitions in that group for it to be causal. If it is strictly ordered after any of those transitions in the zone, then it cannot fire first to satisfy that set; thus, it can never be used in a causal group to set the firing bounds on t_f. The final case is the one presented in Fig. 4.19 extended to members of the required set T' with more than one transition. Transition t_p must be redundant to all transitions in T' to be excluded from the zone; and this only needs to hold for one member of the required set. The intuition follows exactly with the graph in Fig. 4.19(c). If another transition exists that completely determines the firing time of t_f regardless of the presence of t_p, then t_p must be redundant. Every transition in T' must exclude it because an excluding transition must appear in every possible causal group of t_f to make t_p redundant.

The redundant transition definition is demonstrated on an example. Consider the level-ruled Petri net in Fig. 4.4. The zone in the current state is Fig. 4.21(a). The required set for transition t_4 containing only transitions that are found in the zone is $\{\{t_1\}, \{t_3\}, \{t_5, t_7\}, \{t_6, t_7\}\}$. Is transition t_1 a redundant transition for determining when t_4 fires? The first and second cases do not apply. Consider the set $\{t_3\}$ for the third case. The reference point is set to be t_3. The latest time that t_1 can fire after t_3 is the min-max entry (t_1, t_3) in the zone. This is given as -1. The latest firing time of t_4 on t_1 is $\mathsf{Lft}(t_1, t_4) = 5$ and on t_3 is $\mathsf{Lft}(t_3, t_4) = 6$. The first half of the third case in Definition 4.36 is satisfied as $-1 + 5 \leq 6$ holds. The earliest firing time of t_4 on t_1 is $\mathsf{Eft}(t_1, t_4) = 3$ and on t_3 is $\mathsf{Eft}(t_3, t_4) = 2$. The second half of the third case is satisfied as $-1 + 3 \leq 2$ holds; thus, transition t_1 is redundant in this zone. The new zone with t_1 removed is shown in Fig. 4.21(b).

Now consider transition t_3 in the new zone. Is it redundant in setting the firing time of t_4? Again the first two cases do not apply. Consider the set $\{t_6, t_7\}$ for the third case. Transition t_6 is the reference: $\delta_L = -2$ so $-2 + 6 \leq 5$ and $-2 + 2 \leq 3$. Now set t_7 as the reference point: $\delta_L = -1$ so $-1+6 \leq 5$ and $-1+2 \leq 3$. Transition t_3 is redundant in both members of the set, so it can be deleted. The new zone with t_3 removed in shown in Fig. 4.21(c).

Now consider transition t_5 in the new zone. Is it redundant in setting the firing time of t_4? Although the first case in Definition 4.36 does not apply, the second case does. Consider the $\{t_5, t_7\}$ member of the required set for t_4. Transition t_5 cannot fire first with t_7 in this zone—see Definition 4.14 on fire first transitions in z. Transition t_7 always fires before t_5; thus, t_5 can never be used to set the firing bounds on t_4 and can be deleted from z. The final zone with t_5 removed is shown in Fig. 4.21(d). This smaller dimension zone reduces the number of causal assignments

	t_1	t_3	t_6	t_7	t_5
t_1	0	-1	-3	-2	-5
t_3	1	0	-2	-1	-4
t_6	7	6	0	3	-2
t_7	6	5	1	0	-1
t_5	11	10	4	7	0

(a)

	t_3	t_6	t_7	t_5
t_3	0	-2	-1	-4
t_6	6	0	3	-2
t_7	5	1	0	-1
t_5	10	4	7	0

(b)

	t_6	t_7	t_5
t_6	0	3	-2
t_7	1	0	-1
t_5	4	7	0

(c)

	t_6	t_7
t_6	0	3
t_7	1	0

(d)

Fig. 4.21. A series of zones to demonstrate pruning on marking and level satisfied transitions. (a) The initial zone. (b) The zone after t_1 is deleted. (c) The zone after t_3 is deleted. (d) The final zone after t_5 is deleted.

that must be considered by the algorithm to build the finite state space.

4.6.2 Necessary Transitions

In order to prune transitions with rule sets that are only partially marking or level satisfied, the notion of a *necessary transition* must be introduced. A necessary transition is like a required transition only it does not directly affect marking or level satisfied. Consider the net in Fig. 4.22(a). It does not include any syntactic abstraction so the Boolean state is ignored in the following discussion. Transition t_5 is not satisfied by the shown marking. The required set for t_5 consists of the two sets $\{t_3\}$ and $\{t_4\}$. A transition from each of these sets must fire for t_5 to be marking satisfied. The first set, $\{t_3\}$, is already satisfied because t_3 fired to create the current marking. The second set $\{t_4\}$ is not satisfied because t_4 has yet to fire. A *necessary set* for t_5 is a set of transition-delay pairs. The transition in any pair in the set is satisfied by the current marking, and it must fire if t_5 is to be marking satisfied and fire. The delay in any pair in the set is a lower bound on the delay between the transition in the pair and t_5. A necessary set for t_5 is $\{(t_2, 5)\}$. The transition is found by searching backward from t_4 through places that do not appear in the current marking until a transition is found that is satisfied by the current marking. Transition t_2 must fire if t_5 is to be marking satisfied. Transition t_5 cannot fire earlier than 5 time units after t_2.

The necessary set extends to level-ruled Petri nets with syntactic abstraction too because it searches backward on members of the required set that are not satisfied.

Definition 4.37 (Nonsatisfied Set). *The nonsatisfied set for a transition $t \in T$ and a set of fired marking and level transitions $T_\mu, T_\nu \subseteq T$ is any member of the marking and level required set for t that does not contain any transitions from T_μ or T_ν; it is defined as* $\mathsf{nonsatisfied}(t, T_\mu, T_\nu) = \{T' \in \mathsf{mrs}(t) \mid \forall t' \in T_\mu, t' \notin T'\} \cup \{T' \in \mathsf{lrs}(t) \mid \forall t' \in T_\nu, t' \notin T'\}$

The nonsatisfied set for t_5 in the current example is $\{t_4\}$. Consider the net in Fig. 4.22(b). The nonsatisfied set for t_8 depends on the Boolean state of the system. Assume that the current Boolean state is such that both a and c are low. The required set for t_8 is $\{\{t_5\}, \{t_7\}\}$.The nonsatisfied set for t_8 is $\{t_7\}$, since it must fire for a to level satisfy the rule set for t_8 and t_5 has already fired. The necessary set is found by searching backward through members of nonsatisfied sets

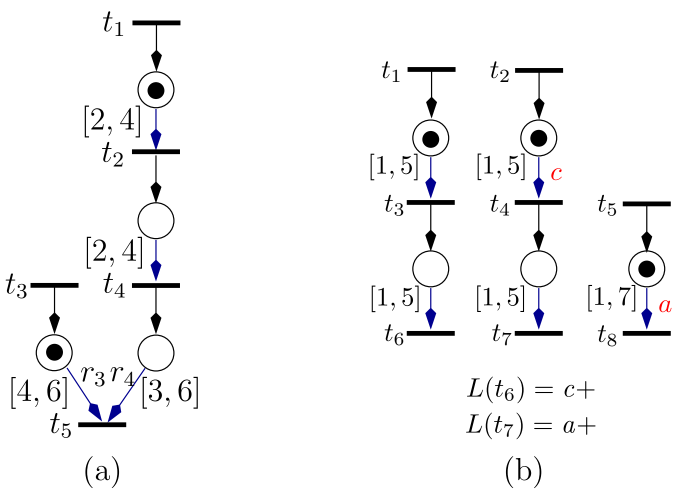


Fig. 4.22. Two nets with transitions that are not enabled. (a) A level-ruled Petri net with no syntactic abstraction where t_5 is not marking satisfied. (b) A level-ruled Petri net with syntactic abstraction where t_8 is not level satisfied.

until t_3 is found. The resulting necessary set is $\{(t_3, 4)\}$. Transition t_8 cannot fire any earlier than 4 time units after t_3.

The necessary set does not always contain a single transition-delay pair. If the nonsatisfied set contains multimember sets, then the necessary set can contain multimember sets too. Consider the level-ruled Petri net in Fig. 4.23(a). It does not use syntactic abstraction, but its required set includes $\{t_1, t_2\}$; thus, there are two backward paths that must be considered in computing the necessary set. The results of both paths are combined at t_3 since it is not known which transition fires in the future to actually create a marking where the rule set for t_3 is satisfied. In either case, however, only the firing of one of the two is needed. Consider now the level-ruled Petri net in Fig. 4.23(b) that uses syntactic abstraction. The required set for t_2 is $\{\{t_1\}, \{t_3, t_4\}\}$. This has a member of its required set with more than one transition, and it has more than one member in its required set. The necessary set now must choose between two results. It can find a result searching backward on $\{t_1\}$, and it can find a result searching backward on $\{t_3, t_4\}$. The two results are not merged because both are required to fire t_2. The result with the smallest cardinality and largest delay is saved at t_2

The algorithm to compute the necessary set is shown in Fig. 4.24. The algorithm has four inputs. The first input, s, is the current timed state class of the level-ruled Petri net. The second parameter, t, is the transition that the necessary set is being

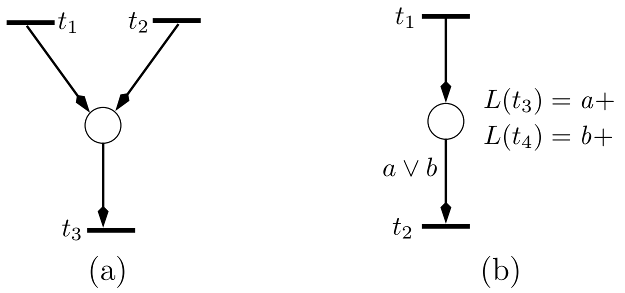


Fig. 4.23. Two level-ruled Petri nets that can generate multimember necessary sets. (a) A level-ruled Petri net with no syntactic abstraction with $\{t_1, t_2\}$ in its marking required set. (b) A level-ruled Petri net with syntactic abstraction with the set $\{t_3, t_4\}$ in its level required set.

```
Algorithm: $\mathsf{necessary}(s, t, T_V, \mathsf{FT}, \mathsf{level})$
 1: /* $s$ and $\mathsf{FT}$ are the three-tuples $(\mu, \nu, z)$ and $(T_\mu, T_\nu, T_z)$ */
 2: if transition $t$ is in the visited transition set $T_V$ then
 3:    return $\emptyset$/* found cycle */
 4: if the rule set of $t$ is satisfied by $(\mu, \nu)$ then
 5:    return $\{(t, 0)\}$
 6: add $t$ to $T_V$
 7: if $\mathsf{level}$ is $\mathsf{true}$ then
 8:    add all $t'$ in $T$ such that $L(t) = L(t')$ to $T_V$
 9: create an empty set of solutions $X$
10: for all transition sets $T'$ in $\mathsf{nonsatisfied}(t, T_\mu, T_\nu)$ do
11:    create empty set $\mathsf{TD}$ of transition-delay pairs
12:    for all transitions $t'$ in $T'$ do
13:       set $\mathsf{level}$ to $\mathsf{true}$ if $t' \bullet \cap \bullet t = \emptyset$ otherwise set it to $\mathsf{false}$
14:       create set $\mathsf{TD}'$ and set it equal to $\mathsf{necessary}(s, t', T_V, \mathsf{FT}, \mathsf{level})$
15:       add $\mathsf{Eft}(t', t)$ to the delay in each member of $\mathsf{TD}'$
16:       union $\mathsf{TD}'$ into $\mathsf{TD}$
17:    add $\mathsf{TD}$ to the solution set $X$
18: get $\mathsf{TD}$ with smallest cardinality and largest delay in solutions $X$
19: return $\mathsf{TD}$
```

Fig. 4.24. An algorithm to compute a set of necessary transitions.

computed for. The third parameter, T_V, is a set of transitions already visited in the search. It is used to terminate the recursion in the algorithm on cycles. The fourth parameter, FT, is the fired three-tuple. It is used to compute the nonsatisfied set. The fifth and final parameter is the Boolean flag level. The flag is used to help terminate the backward search as soon as possible. The return type from the algorithm is a set of transition-delay pairs.

The algorithm is demonstrated on the example in Fig. 4.22. The current state of the system is such that the marking is the one shown and signals a and c are low. The fired three-tuple agrees with the marking and Boolean state in its fired sets. The algorithm is called to compute necessary on t_8.

The algorithm is called as $\mathsf{necessary}(s, t_8, \emptyset, \mathsf{FT}, \mathsf{false})$. It is left empty in this example. Line 2 of the algorithm checks to see if t_8 has already been visited. It has not, so control moves to line 4. It checks to see if t_8 is marking and level satisfied by the current timed state class s. It is not so control moves to line 6. Line 6 updates the visited set with t_8. If t_8 is a transition on a signal and level is set to true, then line 8 adds to T_V all transitions on the same signal in the same direction; thus, the visited set includes knowledge about the signals and uses it to help terminate the backward search as soon as possible.

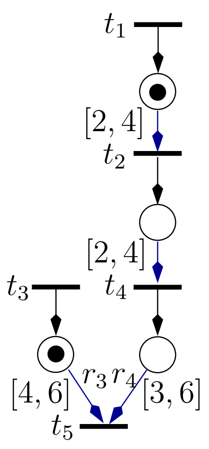
Line 10 iterates on members of the nonsatisfied set for t_8 given the fired marking and Boolean state set from the fired three-tuple. The nonsatisfied set has a single member $\{t_7\}$ in this example. Line 12 iterates on each transition in the current member of the nonsatisfied set. For each member, line 14 makes the recursive call with the new transition and visited set. The results of the recursive call are stored into TD'. This is where the results from each path on a merge place or disjunctive Boolean function are combined. The recursive call is made with $t' = t_7$ and $T_V = \{t_8\}$ in this example.

The recursion continues until the algorithm is called with $t = t_3$ and $T_V = \{t_5, t_7, t_4, t_6\}$. Transition t_3 is yet to be visited and control moves to line 4. Transition t_3 is marking and level satisfied in the current timed state class s; thus, line 5 returns $\{(t_3, 0)\}$. The previous recursive frame gets the new necessary set. It is added to the solution set TD' to give $\{\{(t_3, 0)\}\}$ by line 14. Line 15 adds to the delay in each pair in TD' the earliest firing time between t' and t. The expands the minimum amount of time to elapse from the firing of the enabled transition to the current level of recursion. Line 18 picks the solution with the smallest cardinality. If all of the solutions are empty, then the empty solution is returned. Transition t' is t_3 and t is t_6. The necessary set $\mathsf{TD} = \{(t_3, 1)\}$ is returned to the previous recursive call frame. This continues until recursion is completely unrolled. The final necessary set if $\{(t_3, 4)\}$ for the example.

The necessary set is an important piece to pruning. The goal of pruning is to remove transitions from the zone that no longer contribute useful information. Consider the level-ruled Petri net in Fig. 4.25(a). If it can be shown that t_3 does not contribute useful information to the firing of t_5, then it can be removed from the zone. The necessary set for t_4 is used to determine the usefulness of t_3.

4.6.3 Type II Redundant Transitions

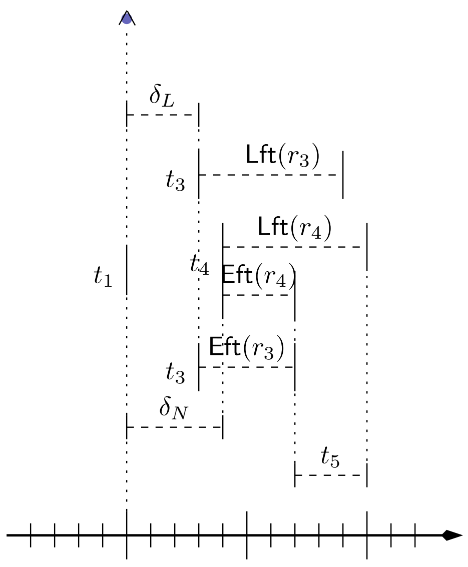
A transition does not have to be marking and level satisfied before pruning on members of its required set begins. Consider again the level-ruled Petri net in Fig. 4.25(a). Transition t_5 is not satisfied by the current marking. Now consider the zone in Fig. 4.25(b). It may be possible to prune t_3 from this zone using knowledge about the structure of the net in Fig. 4.25(a). The first transition that is marking satisfied walking backward from t_4 is t_2. This is a necessary transition for t_5. The earliest that t_4 can fire after t_2 is 2 time units. This is the sum of the lower bounds from t_2 to t_4. Consider the graph in Fig. 4.25(c) showing the earliest and latest firing times of t_5 after t_3 and t_4. The perspective in the graph is relative to t_1. The value δ_N is the earliest time after the necessary transition t_2 that t_5 can fire. This is 4 because the delay from the necessary transition is 2 and the earliest firing time for (t_1, t_2) is 2. The value δ_L is the latest time after t_2 that t_3 can fire as allowed by the zone (i.e., t_1 fires as early as possible in the zone and t_3 fires as late as possible in



(a)

	t_1	t_3
t_1	0	1
t_3	3	0

(b)



(c)

Fig. 4.25. A level-ruled Petri net, zone, and graph to illustrate pruning requirements. (a) The level-ruled Petri net fragment. (b) A zone with separations between t_1 and t_3. (c) A graph showing the earliest placement of t_5 in time depending on its causal assignment.

the zone). Transition t_3 is redundant in the zone to t_5 if $\delta_L + \mathsf{Lft}(r_3) \leq \delta_N + \mathsf{Lft}(r_4)$ and $\delta_L + \mathsf{Eft}(r_3) \leq \delta_N + \mathsf{Eft}(r_4)$; thus, it is guaranteed that the latest firing time is after $\mathsf{Lft}(r_3)$, and the earliest firing time of t_5 is not set by $\mathsf{Eft}(r_3)$. Transition t_3 is redundant in this example because $3 + 6 \leq 4 + 6$ and $3 + 4 \leq 4 + 3$. The graph gives intuition to this relation. The δ_N value is a conservative earliest firing time of t_4 after t_1. The actual firing time of t_4 after t_1 can only be later than δ_N. The δ_L value is the latest time that t_3 can fire after t_1; thus, t_3 can fire only earlier in time in the zone, and t_4 can fire only later in time. This is equivalent to moving t_3 to the left and t_4 to the right in the graph. The earliest and latest firing times are set by some other transition that has yet to fire if the relation holds.

Definition 4.38 (Type II Redundant Transition). *A fired transition $t_p \in T_z$ is redundant to a transition yet to fire $t_f \in T$ in the state $s = (\mu, \nu, z)$, given the fired three-tuple $\mathsf{FT} = (T_\mu, T_\nu, T_z)$, and an initial visited set of visited transitions T_V if the following holds:*

1. *for all $t' \in \mathsf{required\_set}(T_f, T_z, T_z)$, $t_p \notin T'$;*

2. *there exists $T_n \in \mathsf{nonsatisfied}(t_f, T_\mu, T_\nu)$ such that for all transitions $t_n \in T_n$ and for all transition-delay pairs $(t, d) \in \mathsf{necessary}(s, t_n, T_V, \mathsf{FT}, \mathsf{level})$ and for all $t' \in \mathsf{needs}(t) \cap T_z$, $\delta_L + \mathsf{Lft}(t_p, t_f) \leq \delta_N + \mathsf{Lft}(t_n, t_f)$ and $\delta_L + \mathsf{Eft}(t_p, t_f) \leq \delta_N + \mathsf{Eft}(t_n, t_f)$; where* level *is* true *if t_p is connected to t_f through a Boolean function, δ_L is the min-max entry (t_p, t') in z, and δ_N is $d + \mathsf{Eft}(t', t)$.*

Although the definition looks more complex, it is very similar to Definition 4.36 for type I redundant transitions. Transition t_p is made redundant if there exists a member of the nonsatisfied set, T_n, to completely exclude it. This is tested by looking at the necessary set for each member of T_n. If the relation holds for each member, then T_n makes t_p redundant.

A necessary set is a set of transition-delay pairs. The rule set for the transition in a pair in the set is marking and level satisfied, but the transition is not yet fired; thus, it is not yet in the zone. The δ_L value is not known for a transition in a pair in the necessary set. It is, however, known for any transition in the zone needed by a transition in a pair in the set. This is the set $\mathsf{needs}(t) \cap T_z$. It is any transition in the zone that is connected to t by a place or is included in a Boolean function for a rule on t. The bound must be satisfied for each of these transitions. Note that the δ_N value is not known either. The delay d in a pair in the set is the earliest firing time separation between the transition t in the pair and t_f. Recall that t is not yet fired and in the zone; thus, for each transition t' in its $\mathsf{needs}(t) \cap T_z$ set, the value of $\mathsf{Eft}(t', t)$ is added to the delay d. This creates the correct value of δ_N to use in the relation. Everything must satisfy the relation because the actual earliest firing time of t_f is not known and any member of TD may set that time.

4.6.4 Algorithm

The algorithm to prune transitions is shown in Fig. 4.26. The inputs to the algorithm are the current timed state class, s, and the fired three-tuple, FT. The algorithm returns the timed state class with a possibly reduced zone. The algorithm iterates over each transition in the zone. Line 4 iterates over all of the transitions that need t_p. A transition t_f needs transition t_p if t_p is connected to t_f by a place in the level-ruled Petri net, or if t_p is in the level transition set of t_f. Every transition that needs t_p must show it to be redundant to be able to delete t_p from the zone.

```
Algorithm: prune($s$, FT)
 1: /* $s$ and FT are the three-tuples $(\mu, \nu, z)$ and $(T_\mu, T_\nu, T_z)$ */
 2: for all transitions $t_p$ in the transition set $T_z$ do
 3:    set prune to true
 4:    for all transitions $t_f$ that need $t_p$ do
 5:       if $t_p$ is not type I redundant for $t_f$, $z$, and $T_z$ then
 6:          if the rule set of $t_f$ is satisfied by $(\mu, \nu)$ then
 7:             set prune to false and break out of loop
 8:          create initial visited set $T_V$ given $t_p$
 9:          if $t_p$ is not type II redundant for $t_f$, $s$, FT, and $T_V$ then
10:             set prune to false and break out of loop
11:    if prune is true and $t_p$ is not needed in a constraint rule then
12:       delete $t_p$ from the zone $z$ and the zone transition set $T_z$
13: return $s$
```

Fig. 4.26. An algorithm to prune transitions from the timed state class.

The algorithm first tries to show t_p to be type I redundant. A transition t_f that is not marking or level satisfied in its rule set may be excluded as a type I redundant transition. If transitions exist for marking and level satisfied rules in $R(t_f)$ that make t_p type I redundant, then t_p can be removed from the zone. The remaining transitions cover the firing times that may have been set by t_p. If t_p is not type I redundant, then line 6 checks to see if the rule set for t_f is marking and level satisfied. If it is, then t_p cannot be removed from the zone. Line 7 sets the prune flag to false and breaks out of the for loop iterating over transitions that need t_p. If the rule set for t_f is not marking or level satisfied, then t_p may still be type I redundant.

The algorithm to compute the necessary set can affect running time performance of pruning. The initial T_V set is created to terminate the backward search as soon as possible to mitigate its impact.

Definition 4.39 (Initial Visited Set). *The initial visited set given a transition $t_p \in T$ and the current marking and Boolean state (μ, ν) is $T_V = T_w \cup T_r$; where $T_w = \{t \in T \mid (L(t_p) = w+ \vee L(t_p) = w-) \wedge (L(t) = w+ \vee L(t) = w-)\}$, and $T_r = \{t \in T \mid t_p \in \mathsf{needs}(t) \wedge (\mu, \nu) \not\vdash R(t)\}$.*

The first set T_w relies on the correctness definition. A consistent state assignment failure occurs if firing a transition on a signal does not toggle the signal to a new state. The necessary set is used to determine if transition t_p and t can both be causal to some transition t_f. For this to happen, assuming the delays on all rules for t_f are equal, t_p and t must be able to fire first. Transition t_p is already fired and in the zone. Transition t is not marking or level satisfied. If transition t_p is defined on a signal w and in searching for a transition necessary to marking or level satisfying t, the necessary set algorithm encounters another transition on w, then if it is possible to arrive at a state where t_p and t can both be causal, there must be a consistent state assignment failure. This is because t_p and the other transition on w must be concurrent for this to happen, in which case one of the two firing orders leads to a consistent state assignment violation. If the delays on all rules for

t_f are not equal, however, then the delay must be considered before the algorithm can terminate. This is also the case for the second set T_r. If t relies on t_p to fire, then there is a circular dependency. If all delays for rules on t_f are equal, then the necessary set algorithm can terminate because t always arrive after t_p; thus, t_p cannot be causal. If the delays are not equal, however, then the algorithm can terminate if the delay is such that t_p can no longer be causal at that point.

Line 8 creates the initial visited set used to compute the necessary transitions for t_f. Line 9 checks if t_p is type II redundant. If it is not so, then line 10 sets the prune flag to false and breaks out of the loop. If all transitions that require t_p show it to be redundant, then the constraint rules are checked in line 11 before it is pruned. This check is discussed in Section 4.8. If t_p is not needed by a constraint rule, then it is removed from the zone and the set of transitions in the zone by line 12.

4.7 Timing Failures

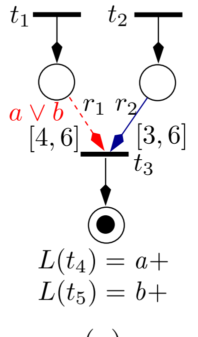
Two types of timing failures on constraint rules can occur in a level-ruled Petri net. The first failure is an earliest firing time failure. This occurs when a transition fires and it has a constraint rule whose timer is below its earliest firing time. The second failure is a latest firing time failure. It relates to any constraint rule of the system. A failure occurs if the timer for any constraint rule that is marking and level satisfied is beyond its latest firing time. These two failures must be checked during the traversal of the reachable state space.

An earliest firing time violation can be detected by looking at the minimum separation between a clock on a constraint rule being reset and the firing of its transition. If the minimum separation is below its earliest firing time, then there is a failure.

Definition 4.40 (Earliest Firing Time Constraint Failure). *The zone z contains an earliest firing time failure for a constraint rule $r \in C$ given the set of transitions in the zone $T_z \subseteq T$ and a reference transition $t_f \in T$ if r is a constraint rule for t_f and $\phi_{\mathsf{min}}(r, z) < \mathsf{Eft}(r)$; where $\phi_{\mathsf{min}}(r, z) = \min(\bigcup_{T_r \in \mathcal{T}} \phi_{\mathsf{max}}(T_r, z))$ is the min-*

imum separation between the firing of t_f and each member of the required set $\mathcal{T} =$ $\mathsf{required\_set}(r, T_z, T_z)$ *being satisfied;* $\phi_{\mathsf{max}}(T_r, z) = \max(\bigcup_{t_r \in T_r} -1 \cdot z_{\min}(t, t_f))$ *is the maximum separation between the firing of t_f and T_r being satisfied; and $z_{\min}(t, t_f)$ is the min-max entry for (t, t_f) in the zone z; the function* $\mathsf{Eft\_failure}(r, t_f, z, T_z)$ *returns the failure if it exists.*

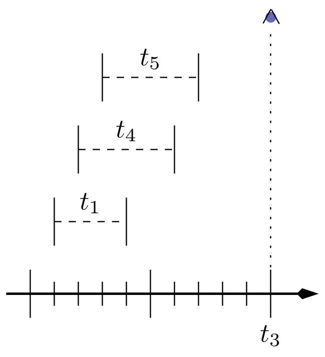
The definition is best understood through example. Consider the level-ruled Petri net in Fig. 4.27(a). The marking is the one shown. The Boolean state is such that both a and b are high. The current zone is given in Fig. 4.27(b). The rule r_1 is a constraint rule, while r_2 is an ordinary rule. The zone shows that t_3 is the last transition to fire. The goal is to see if there is an earliest firing time violation on any firing time of t_3 allowed by the zone. Consider the graph in Fig. 4.27(c). The firing time of t_3 is the fixed reference point in the graph. The graph shows the times at which transitions t_1, t_4, and t_5 fire relative to the reference point of t_3, and it is drawn to match the separations in the zone in Fig. 4.27(b). The minimum separation between t_5 and t_3 is the min-max entry on (t_5, t_3) in the zone, which is -3; it means that t_5 fires as little as 3 time units before t_3 as shown in the graph. The maximum separation between t_5 and t_3 is the max-min entry on (t_3, t_5) in the zone, which is 7. It means that t_3 can fire as much as 7 time units after t_5 as drawn



(a)

	t_1	t_2	t_4	t_5	t_3
t_1	0	-1	0	-1	-6
t_2	3	0	2	1	-3
t_4	2	0	0	0	-4
t_5	3	1	2	0	-3
t_3	9	6	8	7	0

(b)



(c)

Fig. 4.27. A level-ruled Petri net, graph, and zone to illustrate a timing failure. (a) A level-ruled Petri net with a constraint rule on t_3. (b) The zone in the current state. (c) A graph showing the possible firing times of t_1, t_2, t_4, and t_5 relative to t_3.

in the graph. The other graph separations are derived similarly.

Definition 4.40 computes the minimum allowed separation between the constraint rule on r_1 being reset and the firing of t_3. It looks at each member of the required set individually to compute this. The required set for r_1, containing only transitions in the zone, is $\{\{t_1\}, \{t_4, t_5\}\}$. The minimum separation between the set $\{t_1\}$ being satisfied and the firing if t_1 is $\phi_{\mathsf{max}}(\{t_1\}, z) = 6$ as shown in the graph. The minimum separation for the set $\{t_4, t_5\}$ being satisfied and the firing of t_3 depends on which transition in the set fires first. Although t_5 fires 3 time units before t_3, it can never be causal at that point because t_4 cannot fire any later than 4 time units before t_3; thus, the separation between this set being satisfied and t_3 firing is $\phi_{\mathsf{max}}(\{t_4, t_5\}, z) = 4$. This is latest time at which either one of the two transitions can fire first before t_3. The clock for the constraint rule cannot reset before t_1 fires and either t_4 or t_5 fires. The reset point defines the minimum separation that can exist between r_1 resetting and t_3 firing. Although the set $\{t_1\}$ is satisfied 6 time units before t_3, the resetting of the r_1 clock can be delayed by the set $\{t_4, t_5\}$ until 4 time units before t_3; thus, the minimum separation between the resetting of the r_1 clock and the firing of t_3 is $\phi_{\mathsf{min}}(r_1, z) = 4$. If this minimum separation is below the earliest firing time for the rule, then there is a failure. This is not the case in this example as $4 \geq \mathsf{Eft}(r_1)$.

A latest firing time violation can be detected by looking at the maximum separation between a clock on a constraint rule being reset and a reference transition. If the maximum separation is above the latest firing time of the rule, then there is a failure. Note that the reference transition does not need to be the transition checked by the constraint rule. A failure can occur if any marking and level satisfied constraint rule has a timer outside of the latest firing time on the rule; thus, the most recently fired transition in the zone is always used as a reference point.

Definition 4.41 (Latest Firing Time Constraint Failure). *The zone z contains a latest firing time failure for a constraint rule $r \in C$ given the set of transitions in the zone T_z and a reference transition t_f if $\theta_{\mathsf{min}}(r, z) > \mathsf{Lft}(r)$; where $\theta_{\mathsf{min}}(r, z) = \min(\bigcup_{T_r \in \mathcal{T}} \theta_{\mathsf{max}}(T_r, z))$ is the minimum separation between the firing of*

t_f and each member of the required set $\mathcal{T} = \mathsf{required\_set}(r, T_z, T_z)$ being satisfied; $\theta_{\mathsf{max}}(T_r, z) = \max(\bigcup_{t_r \in T_r} z_{\max}(t_f, t))$ is the maximum separation between the firing of t_f and T_r being satisfied; and $z_{\max}(t_f, t)$ is the max-min entry for (t_f, t) in the z; $\mathsf{Lft\_failure}(r, t_f, z, T_z)$ returns the failure if it exists.

Definition 4.41 computes the maximum allowed separation between the timer on the constraint rule for r_1 being reset and the firing of t_3. It looks at each member of the required set individually to compute this like Definition 4.40. The maximum separation between the set $\{t_1\}$ being satisfied and the firing if t_1 is $\theta_{\mathsf{max}}(\{t_1\}, z) = 9$ as shown in the graph. The maximum separation for the set $\{t_4, t_5\}$ being satisfied and the firing of t_3 depends on which transition in the set fires first. Although t_5 fires 7 time units before t_3, t_4 can be causal even earlier because it fires 8 time units before t_3; thus, the maximum separation between this set being satisfied and t_3 firing is $\theta_{\mathsf{max}}(\{t_4, t_5\}, z) = 8$. This is the earliest time that one of the two transitions can fire before t_3. The clock for the constraint rule cannot reset before t_1 fires and either t_4 or t_5 fires. This point defines the maximum separation that can exist between the r_1 clock resetting and t_3 firing. Although the set $\{t_1\}$ is satisfied 9 time units before t_3, the resetting of the r_1 clock can be delayed by the set $\{t_4, t_5\}$ until 8 time units before t_4; thus, the maximum separation between the resetting of the r_1 clock and the firing of t_3 is $\theta_{\mathsf{min}}(r_1, z) = 8$. If this maximum separation is above the latest firing time for the rule, then there is a failure. This is the case in this example as $8 \geq \mathsf{Lft}(r_1)$.

The algorithm to check timing failures is shown in Fig. 4.28. Line 2 of the algorithm checks for earliest firing time violations in constraint rules associated with the fired transitions. Line 4 reports an earliest firing time failure if any occur. Line 5 checks for latest firing time failures. Line 6 restricts the check to constraint rules that are marking and level satisfied. Line 8 reports a latest firing time failure if one exists. Line 9 returns no failure if no timing violations exist in the current state.

```
Algorithm: timed_failure($t_f, s, \mathsf{FT}$)
1: /* $s$ and FT are the three-tuples $(\mu, \nu, z)$ and $(T_\mu, T_\nu, T_z)$ */
2: for all rules $r$ in $C(t_f)$ do
3:   if Eft_failure($r, t_f, z, T_z$) then
4:     return earliest firing time violation on $r$
5: for all rules $r$ in $C$ do
6:   if $r$ is satisfied by $(\mu, \nu)$ then
7:     if Lft_failure($r, t_f, z, T_z$) then
8:       return latest firing time violation on $r$
9: return no timing failure
```

Fig. 4.28. An algorithm to check timing failures on constraint rules.

4.8 Constraint Redundant

Pruning must pay special attention to transitions required by constraint rules. These cannot be pruned using the type I and type II redundant properties.

Definition 4.42. *Transition $t_p \in T$ is redundant to the constraint rule $r \in C$ in the zone z from the current state $s = (\mu, \nu, z)$ given the set of transitions in the zone T_z if*

1. *for all members $T_r \in \mathsf{required\_set}(r, T_z, T_z)$ such that $t_p \in T_r$, $\theta_{\mathsf{max}}(T_r, z) \neq z_{\max}(t_f, t_p)$, and $\phi_{\mathsf{max}}(T_r, z) \neq z_{\min}(t_p, t_f)$; and*
2. *there exists a member $T_r \in \mathsf{required\_set}(r, T_z, T_z)$ such that $t_p \notin T_r$, $\theta_{\mathsf{min}}(r, z) \neq z_{\max}(t_f, t_p)$, and $\phi_{\mathsf{min}}(r, z) \neq z_{\min}(t_p, t_f)$.*

where t_f is the last transition in the zone (i.e., the last transition that fired to create the zone); t_p is redundant if for all constraint rules $r \in C$ such that $t_p \in \mathsf{needs}(r)$, $(\mu, \nu) \vdash r$ and t_p is redundant to r.

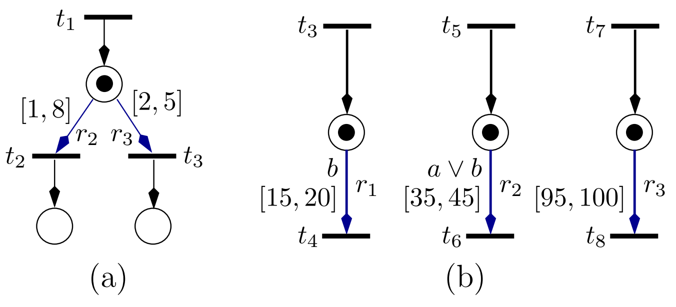
A transition is redundant if it is not the transition that defines the minimum and maximum separation between the resetting of the constraint rule clock and the firing of t_f. This falls directly from the earliest and latest firing time failure definitions. Consider the level-ruled Petri net in Fig. 4.27(a) with its zone in Fig. 4.27(b). Suppose that there exists another constraint rule r_3 that is identical to r_1, only it is defined for a different transition (i.e., t_1 feeds its place, but its place is not connected to t_3. Transition t_3 is the last transition to fire in the zone; thus, it is the reference point. Transition t_1 is redundant because it is not used to set the minimum and maximum separation between the resetting of the r_3 clock and t_3. It can be removed from the zone.

4.9 Exactness

Unfortunately, the timing analysis algorithm is not exact in building a finite representation of the reachable timed state space. It is, however, conservative. It is possible to include behaviors that do not actually exist in the net. There are two factors that contribute to the analysis algorithm being conservative. The first is not ordering transitions involved in choice constructs. The second is not adding all marking and level satisfied transitions to the zone. This section illustrates the two scenarios. Their impact is evaluated in Chapter 6.

A choice construct can introduce extra behaviors into the finite representation. Consider the level-ruled Petri net fragment in Fig. 4.29(a). The transitions t_2 and t_3 are both fireable in the current marking regardless of the Boolean state. The timers for the rules r_2 and r_3 are reset at the firing of t_1. The semantics of the level-ruled Petri net never allow t_2 to fire at the latest firing time of r_2. This is because the latest firing time r_3 is below the latest firing time of r_2. Transition t_3 must fire when its timer reaches 5 according to the level-ruled Petri net semantics; thus, t_2 must also fire before its timer reaches 5; otherwise it cannot fire at all because the firing of t_3 disables it. The timing analysis algorithm lets transition t_2 fire at the latest firing time for r_2; thus, transition t_2 can fire as much as 8 time units after t_1 in the finite representation. This is because t_2 and t_3 only exist in the zone together long enough to see if they are concurrently fireable. After the fireable set is known, then each transition is fired individually from the fired set without considering the other enabled transitions. This over approximation can lead to false negatives in the verification process and may create nonexact circuits in a state based synthesis algorithm. This has only been seen in contrived examples designed to exploit the weakness in the algorithm.

Ignoring other marking and level satisfied transitions in computing successors timed state classes can add extra behaviors into the finite representation. This is again seen in computing fireable transitions from the current timed state class. Consider again the level-ruled Petri net in Fig. 4.29(b) with the current zone in Fig. 4.29(c). There is a single causal assignment for t_4 and a single causal assignment for t_8. There are two causal assignments, however, for t_6. The two causal assigned zones that are considered by the fireable algorithm are shown in Fig. 4.30. The zone in Fig. 4.30(a) for the t_1 causal assignment to t_6 only allows t_4 and t_8 to be fireable. The zone in Fig. 4.30(b) for the t_2 causal assignment to t_6, however, allows t_4, t_6, and t_8 to be fireable. Transition t_6 is only fireable on one of its two causal assignments when all the marking and level satisfied transitions with their causal assignments are considered. The successor function, however, does not use this information. At the firing of t_6 in the depth-first search from this state, the algorithm to compute successor states creates a successor for both causal assignments to t_6 and adds both assignments to the state space. There are no future states from the t_1 causal assignment because it does not allow t_6 to fire before t_4 and t_8; thus, the causal assignment to all marking and level satisfied transitions affects the allowed separations for the transition being fired in the depth-first search. Ignoring this information allows extra behavior into the finite representation of



$$\begin{array}{c|ccc} & t_7 & t_1 & t_2 \\ \hline t_7 & 0 & -60 & -60 \\ t_1 & 75 & 0 & 15 \\ t_2 & 75 & 15 & 0 \end{array}$$

$L(t_1) = a+$
$L(t_2) = b+$

(c)

Fig. 4.29. Two level-ruled Petri nets with a zone that create conservative zones in the reachable state space. (a) A simple choice construct with different delay bounds for t_2 and t_3. (b) A causal assignment that leads to no fireable transitions. (c) The zone for the current state of the net in (b).

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	-75	-95	-95
t_1	75	0	15	0	-20	-20
t_2	75	0	0	-15	-35	-20
t_4	95	20	20	0	-15	0
t_6	120	45	45	30	0	25
t_8	100	40	40	25	5	0

(a)

	t_7	t_1	t_2	t_4	t_6	t_8
t_7	0	-60	-60	-75	-95	-95
t_1	75	0	0	-15	-35	-20
t_2	75	15	0	-15	-20	-20
t_4	95	35	20	0	0	0
t_6	120	45	45	30	0	25
t_8	100	40	40	25	5	0

(b)

Fig. 4.30. Two causal assigned zones in their canonical form showing fireable transitions. (a) The causal assignment of t_1 to t_6 showing only t_4 and t_8 as fireable. (b) The causal assignment of t_2 to t_6 showing only t_4, t_6, and t_8 as fireable.

the state space. This case has been seen in real examples, but it occurs very infrequently.

4.10 Related Work

The addition of time to any method of state space exploration, in general, exacerbates state explosion. This is because any approach must define a timed state that not only describes the *untimed state* of the system (i.e., the value of all signals in the circuit) but information indicating the occurrence of that untimed state in time. A set of timed states may have a common untimed state but be unique in their occurrence in time; thus, the time representation in the timed state now affects the size of the timed state space. A discrete model of time divides time into a minimum discrete quantum. This bounds the size of the timed state space because time is no longer a continuous quantity [59]. This approach lends itself to symbolic methods to compactly represent the timed state space [20, 60]. Practical results using discrete methods are promising as recently shown by [26, 21]. Recently, these techniques are applied to timed circuit analysis in [28]. Choosing an adequate time quantum, however, is critical to discrete timing analysis. If the quantum is too large, then behaviors of the system are masked and lost. If the quantum is too small, then the timed state space is too large to manage [19]. Recent work addresses some of these issues by combining dense and discrete time semantics in [27]. The dependence on the size of the time quantum in discrete models erodes confidence in the analysis of complex timing constraints. A masked timing behavior due to the size of the time quantum can lead to an actual timing failure in an aggressively timed circuit. A more precise time model is required for circuit analysis.

A continuous time model does not restrict when things occur. In a continuous time model, timing information is often represented by *time separations*—the amount of time that elapses between any two transitions in the circuit. A set of seemingly concurrent transitions in a circuit may actually be ordered in time due to various combinations of delays. A timed state space can be derived by ordering these seemingly concurrent transitions according to their time separations. The minimum and maximum time separations of transitions in the circuit can be calculated from the structure of the circuit model. These minimum and maximum separations are used to approximate the actual timed state space [42, 61]. Algorithms to calculate the minimum and maximum time separations do not adequately address nondeterminism in environment models, as well as classes of circuits that include arbiters.

Non-determinism can be analyzed in a continuous time model by deriving time separations from execution traces in the circuit. Each explored trace can potentially generate a unique set of time separations at each untimed state of the trace. A set of time separations at a given untimed state can be grouped into equivalence classes called regions [52, 62]. A timed state is created for each unique region generated at an untimed state during the trace evolution. A region naturally addresses the continuous nature of the timed state space but is too small to significantly reduce the number timed states. Zones extend regions by representing larger equivalence classes using convex hulls [63]. These can be represented in *difference bound matrices* (DBMs) since they represent allowed separations in a system [63], and many efficient algorithms have been developed to manipulate DBMs. DBMs can also be implicitly represented [27]. There have been several published algorithms using zones for timing analysis. A tool for timed automata is presented in [23]. A tool for time Petri nets is presented in [51]. These tools provide a total order analysis of both system models, as well as a partial order reduction in transitions firing which is discussed in the next chapter. Other interesting work in the analysis of timed automata is in [64].

A zone represents allowed execution traces in the circuit. A different ordering on a set of concurrent transitions, however, can lead to a different zone. This creates an exponential branching in the timed state space. As the zone is a partially ordered set relating various transition times, it is possible to reduce branching in the timed state space by adding fewer relations to the set. *Local time semantics* and *partially ordered sets* (POSETs) remove orderings in the zone on sets of independent concurrent signal transitions reducing the representation size of the timed state space [10, 24, 25, 57, 65, 66].

POSET reduction in the timing analysis of *Timed Petri nets* (TPN)—Petri nets with timers on the places—yields a significant reduction in the number of zones associated with each marking [66]. The algorithm in [66] implements the POSET reduction on concurrent independent signal transitions. It explicitly models, however, timers on the places; thus, it needlessly considers many redundant orderings of firings of these timers [25]. More importantly, it cannot address arbitrary Boolean functions in the syntactic abstraction.

4.11 Summary

This chapter details a timing analysis algorithm for level-ruled Petri nets. The algorithm supports arbitrary Boolean expressions in the syntactic abstraction, and it implements a partial order reduction in the timing information. This chapter also presents a method to conservatively prune information from the timed state class representation to further reduce the cost of timing analysis. This method uses a recursive algorithm to search backward in the level-ruled Petri net for necessary transitions that must fire to enable transitions that are partially marking satisfied or not yet level satisfied.

The timing analysis algorithm includes methods to validate the correctness properties. The validation occurs in two separate steps. The first step checked untimed properties such as safety, consistent state assignment, output semimodular, and untimed properties of constraint rules. The second step checks timed properties in constraint rules. The correctness validation has minimal impact on the cost of timing analysis.

The timing analysis algorithm as presented is not exact in building a finite representation of the reachable state space. It has two sources of inexactness. The first is a result of allowing conflicting transitions to fire later than allowed by the level-ruled Petri net semantics. A false negative resulting from this inexactness has only been seen in contrived examples. The second source of inexactness is in how the fireable set is constructed. Although this has been observed in practical examples, it occurs very infrequently and rarely leads to a false negative result.

CHAPTER 5

REDUCTION

The timed state space of a level-ruled Petri net is often too large to be managed. This is the case even in seemingly trivial designs. As a result, timing analysis can only be applied to a constrained set of circuits. The analysis is often not direct because the circuits are simplified to again manage the explosion in the number of reachable timed states. This is a significant impediment to methods that rely on timed state space traversal for analysis.

The goal of modular analysis is to examine a single module in a larger system. This is accomplished by only exploring the portion of the actual reachable state space that is of interest to the module being analyzed; thus, complete analysis of the system requires the separate examination of every module in the system. The assumption is that the analysis cost of each subcircuit is very small. The cumulative cost of analyzing each module in the system is then small compared to a complete flat analysis.

Consider the network of level-ruled Petri nets $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$ of modules describing a timed circuit. The parallel composition of the network, M, is closed, and $\mathcal{E}$ is a pure environment model for the circuit. Recall that a level-ruled Petri net is the tuple from Definition 3.3 that is $M = (N, E, C)$, where $N = (T, P, F, \mu_o)$ is an ordinary Petri net, $E = (W, L, \nu_o, \mathsf{Eft}, \mathsf{Lft}, \mathsf{Lsat})$ is a level-ruled extension for the net N, and C is a set of constraint rules. The module M_i is the level-ruled Petri net specification of the i-th subcircuit; thus, $M_i = (N_i, E_i, C_i)$. The goal of modular analysis is to consider only module M_i in the system M; thus, all modules in the parallel composition of the network, excluding M_i, become environment models to M_i. Any analysis results of M_i apply only to the given system M. If a circuit is synthesized from the timed state space of M_i in the system M and shown to be correct, then it may work incorrectly in a different module set. The modular analysis procedure depends on the behavior of the other modules $\mathcal{E}$, M_1, …, M_n to define the input behavior to M_i. Analysis results for the subcircuit in this system do not apply to the same subcircuit in other systems.

This chapter first presents a partial order reduction for the level-ruled Petri net. The partial order reduction explores only firing orders during state space traversal that are relevant to showing correctness in a module. The chapter then extends the partial order reduction to not only verify correctness in a module, but to generate the complete reachable state space of the module as allowed by the larger system in which it is found. The end goal of this analysis is synthesis. A timed circuit synthesis algorithm is presented in [42]. The algorithm requires the complete state space of the timed circuit. Modular synthesis generates this state space in the environment in which the circuit is found. The approach expands the set of examples for which timed circuit synthesis can be applied because the analysis no longer needs to enumerate the entire state space in which a module is found. It only needs to enumerate the states in the entire state space that are visible to the module proper.

5.1 Partial Order Reduction

The idea of partial order reduction is to avoid exploring all successor timed state classes during state space exploration. This is accomplished by omitting possible firing orders on fireable transitions. The omitted firing orders must be carefully chosen to not hide key behaviors in the target submodule. This section defines the basis of the partial order reduction algorithm and its various support sets. The presentation in this section closely follows the presentation by Minea in [25]. Although Minea did not originate the theory of partial order reduction, he did unify much of the research in partial order reduction into a general framework. More importantly, Minea proposes a partial order reduction for timed event/level structures in [25]. Timed event/level structures are very similar to the level-ruled Petri net. The presentation by Minea draws heavily on work by Valmari in [67] and Godefroid in [68] for the untimed theory in the partial order reduction. Minea follows Yoneda's work in [56] for the theory relating to timing information in the partial order reduction for timed event/level structures. The same notation and names are used in this presentation as are used by Minea and Yoneda to help the reader relate to the other works.

5.1.1 Basic Notions

There are two basic notions that are important to this presentation of partial order reduction: an execution sequence and enabled transitions. These are not new notions in the level-ruled Petri net semantics, but they are cast to follow published literature relating to partial order reduction.

Definition 5.1 (Execution Sequence). *An execution sequence is a pair of vectors* $(\mathbf{s}, \mathbf{t})$ *where* $\mathbf{s} = (s_0, s_1, s_2, \ldots, s_n)$, $\mathbf{t} = (t_1, t_2, \ldots, t_n)$, *and* $s_i = (\mu_i, \nu_i, z_i)$ *with the following properties: for all* $i \in \mathbb{N}$ *such that* $1 \leq i \leq n$, $(\mu_{i-1}, \nu_{i-1}) \vdash t_i$, *and there exists a casual assignment to* t_i *such that firing* t_i *in* s_{i-1} *leads to* s_i*; the set of timed state class sequences reachable from an initial timed state class of a level-ruled Petri net* M *is* $\mathcal{F}(M)$ *(i.e.,* s_0 *is the initial state of the system); the notation* $(s, t, s') \in (\mathbf{s}, \mathbf{t})$ *indicates that there exists an* $i \in \mathbb{N}$ *such that* $1 \leq i \leq n$, $s_{i-1} = s$, $t_i = t$, *and* $s_i = s'$*; the notation* $(s, t, s') \in \mathcal{F}(M)$ *indicates that there exists a sequence* $(\mathbf{s}, \mathbf{t})$ *in the set* $\mathcal{F}(M)$ *such that* $(s, t, s') \in (\mathbf{s}, \mathbf{t})$ *holds.*

An execution sequence in a level-ruled Petri net relates to a firing sequence. The set of reachable execution sequences from the initial timed state class of the level-ruled Petri net M is constructed during timing analysis.

The set of all reachable timed state classes of a level-ruled Petri net is given by the set $S(M)$. This too is constructed during state space exploration.

Definition 5.2 (Enabled Transitions). *A transition* $t \in T$ *is enabled in a timed state class* $s \in S(M)$ *if there exists a timed state class* s' *such that* $(s, t, s') \in \mathcal{F}(M)$*;*

the set of all enabled transitions in s is $\mathsf{enabled}(s) = \{t \in T \mid t \textit{ is enabled in } s\}$.

An enabled transition in a timed state class s is fireable from s. A transition that is not enabled in s is said to be *disabled.*

5.1.2 Basic Principles

There are two basic principles in partial order reduction: independence and visible. The traditional definition of independence only considers two properties. Enabledness and commutativity. This definition introduces a new property, correctness, to deal with the correctness properties in the level-ruled Petri net semantics.

Definition 5.3 (Independent Transitions). *Two transitions $t, t' \in T$ are independent in the timed state class $s \in S$ if they satisfy the following conditions:*

1. ***Enabledness**:* $t \in \mathsf{enabled}(s) \implies (t' \in \mathsf{enabled}(s) \iff \forall s' \in S : (s, t, s') \in \mathcal{F}(M),\ t' \in \mathsf{enabled}(s'))$, *and symmetrically,* $t' \in \mathsf{enabled}(s) \implies (t \in \mathsf{enabled}(s) \iff \forall s' \in S : (s, t', s') \in \mathcal{F}(M),\ t \in \mathsf{enabled}(s'))$,

2. ***Commutativity**:* $t, t' \in \mathsf{enabled}(s) \implies \forall s_i, s_j, s_x, s_y \in S : (s, t, s_i, t', s_j) \in \mathcal{F}(M) \wedge (s, t', s_x, t, s_y) \in \mathcal{F}(M),\ s_j = s_y$, *where the notation* $(s, t, s_i, t', s_j) \in \mathcal{F}(M)$ *is equivalent to* $(s, t, s_i) \in \mathcal{F}(M)$ *and* $(s_i, t', s_j) \in \mathcal{F}(M)$;

3. ***Correctness**:* $t, t' \in \mathsf{enabled}(s) \implies \forall s_i, s_j, s_x, s_y \in S : (s, t, s_i, t', s_j) \in \mathcal{F}(M) \wedge (s, t', s_x, t, s_y) \in \mathcal{F}(M)$, $(\mathbf{s}, \mathbf{t})$ *and* $(\mathbf{s}', \mathbf{t}')$ *are safe, consistent, output semimodular, and constraint satisfied execution sequences for a specified module M_i in the system M or both are failures of the same, where* $\mathbf{s} = (s, s_i, s_j)$, $\mathbf{t} = (t, t')$, $\mathbf{s}' = (s, s_x, s_y)$, *and* $\mathbf{t} = (t', t)$

The definition of independence relies on three conditions. The first two conditions are common to partial order work. The first condition relates to Enabledness. Two independent transitions cannot contribute to the enabledness of each other. This means that the firing of the one does not disable the other; it also means that the firing of one does not enable the other.

The second condition relates to commutativity. If the two transitions are enabled in the current timed state class, then there exists from that state execution sequences where the transitions fire in either order. Commutativity states that in any future state of the execution sequence where the two transitions fire adjacent to each other, firing them in any order leads to the same future state. The POSET method makes commutativity possible. It unorders transitions in the zone creating the same zone regardless of the firing order. In this sense, there exists an order of transition firings that creates a zone that is a superset of all other zones.

The third condition relates to correctness. The correctness properties in the level-ruled Petri net semantics are best addressed through independence in the partial order reduction because it is not based on a temporal specification. This is a similar approach taken by Yoneda in [56] for a related set of correctness properties in the time Petri net. The correctness condition requires that both firing orders of the two enabled transitions result in a correct execution sequence. A correct execution sequence is one that is safe, consistent state assigned, output semimodular in all modules but the environment, and constraint satisfied with respect to a target submodule that is being verified; thus, the two enabled transitions cannot enable or disable one another, and firing them in either order does not lead to a failure condition in the level-ruled Petri net semantics too. The definition of a safe, consistent state assigned, output semimodular in all but the environment, and constraint satisfied execution sequence is the natural extension of Definition 3.8, Definition 3.11, Definition 3.15, and Definition 3.17 on firing sequences to execution sequences respectively.

The next basic notion of partial order reduction is visible. Visible traditionally relates to transitions in a formula consisting of atomic propositions. The formula represents a specification of properties for a module in the system, and the timed state space of the module is checked to see if it violates the properties in the specification. The atomic propositions in this application are signal values and time differences between transitions. Note, that the notion of visible can be used to verify next-time free linear temporal logic. A transition t is *invisible* with respect to a set of atomic propositions appearing in any formula in the specification of a module if in any two states s,$s' \in S$ connected by t, the labeling of the two states is identical with respect to the set of atomic propositions in the formula. A signal that is not invisible is visible. Although this is beyond the scope of verification in this work, there is nothing that precludes the use of next-time free linear temporal logic to validate properties in the level-ruled Petri net model of a system. The visible set is used, however, in this application to implement modular synthesis using partial order reduction.

5.1.3 Conditions

Partial order reduction traverses a subset of the total reachable state space to validate correctness. It does this by selecting a reduced set of enabled transitions to explore at a given state. This reduced set of transitions is the ample set $\mathsf{ample}(s)$. The smallest ample set at any given state yields the largest reduction. The algorithm is always safe when $\mathsf{ample}(s) = \mathsf{enabled}(s)$. This section describes conditions of the ample set for the partial order reduction to be safe—meaning that if any failure condition for a module M_i exists in the execution sequence set $\mathcal{F}(M)$ of the system M, than a failure condition for M_i also exists in the reduced set of the execution sequences $\mathcal{F}_{M_i}(M) \subseteq \mathcal{F}(M)$ explored by the partial order reduction using ample sets.

The ample set conditions are presented as if the complete reachable set of execution sequences in known. They are properties that the set must have for the reduction to be safe. An algorithm to construct the ample set during state space exploration is presented in the next section.

Property 5.1 (Emptiness). $\mathsf{ample}(s) = \emptyset \iff \mathsf{enabled}(s) = \emptyset$

An ample set contains no transitions at a given state if and only if there are no successor transitions from that state in any reachable execution sequence of the system.

Property 5.2 (Faithful Decomposition). *For any portion,* $(\mathbf{s}, \mathbf{t})$*, of an execu-*

tion sequence in $\mathcal{F}(M)$ and for any index $k \in \mathbb{N}$ such that $k \leq [\mathbf{t}]$, if $t_j \notin \mathsf{ample}(s_0)$ for $0 < j \leq k$, then t_j is independent of any transition $t \in \mathsf{ample}(s_0)$ for $0 < j \leq k$.

Faithful Decomposition means that any transition not in $\mathsf{ample}(s)$ is independent of the transitions in $\mathsf{ample}(s)$; thus, the firing of any independent transition cannot affect the execution of any transition in $\mathsf{ample}(s)$. The ample set effectively divides the enabled transitions at a state into a dependent set, which is the ample set, and an independent set, which is the enabled set minus the ample set. Note that the definition of independence in transitions is critical to the verification of correctness properties in the level-ruled Petri net semantics. Transitions cannot disable or enable each other. Moreover, they must satisfy the correctness requirement in that firing them in any order always leads to the same failure or no failure at all.

Property 5.3 (Visibility). *If the set $\mathsf{ample}(s)$ contains a visible transition, then $\mathsf{ample}(s) = \mathsf{enabled}(s)$.*

The visibility condition relates to the construction of the ample set. The initial ample set begins with some invisible transition in $\mathsf{enabled}(s)$. Transitions that are dependent with the initial invisible transition are then added until a fixed point is reached on the ample set. There are two scenarios that can occur: first, there are no invisible transitions in $\mathsf{enabled}(s)$; thus, $\mathsf{ample}(s) = \mathsf{enabled}(s)$, since all visible transitions affect the correctness of the target submodule and their various firing orders are important. Second, every invisible transition in $\mathsf{enabled}(s)$ depends on a visible transition in $\mathsf{enabled}(s)$; thus, $\mathsf{ample}(s) = \mathsf{enabled}(s)$ by transitivity.

Property 5.4 (Cycle Closing). *A transition that is enabled in every state of a cycle in the reduced set $\mathcal{F}_{M_i}(M)$ belongs to the ample set of some state on the cycle.*

A cycle is a sequence of states that is repeated in the system. Cycle closing is the final property of the ample set for the reduction to be safe. This condition prevents transitions from being ignored in cycles and guarantees that all behaviors important to the correctness of the module are explored. The condition must dynamically detected in the state space traversal.

These four conditions are sufficient to guarantee that the partial order reduction does not miss a failure in a target submodule during state space exploration if one exists. The conditions also provide a general framework for designing algorithms to dynamically construct the ample set at each stage of the state space traversal. This is the topic of the next subsection.

5.1.4 Ample Set Construction

The construction of the ample set at a given timed state class of the state space traversal relies on the definition of dependence and visibility in transitions. These must be defined before the algorithm to construct the ample sets is presented. If the firing of one transition disables the other transition, then the two transitions are dependent. There are two ways in the level-ruled Petri net semantics to disable an enabled transition: through a change in marking and through a change in Boolean state.

A transition is disabled through a change in the marking if a token in its preset is removed by the firing of another transition. Consider the level-ruled Petri net in Fig. 5.1(a). The firing of t_1 disables t_2 because it removes from the marking the place in the shared preset. This is a disabling through a change in marking. Both firing orders of t_1 and t_2 must be explored in the partial order reduction if they exist in the full state space of the system or again failures can be missed.

A transition is disabled through a change in Boolean state if the firing of another transition causes a rule in its rule set to no longer be level satisfied by the new Boolean state. Consider the level-ruled Petri net in Fig. 5.1(b). The rule for t_2 requires the signal a to be high to be level satisfied in the Boolean state. The transition t_1 is mapped to $a-$ moving this signal into a low state. The firing of t_1 can disable t_2. This is a disabling through a change in Boolean state. Both firing orders of t_1 and t_2 must be explored in the partial order reduction if they exist in the full state space of the system or failures can be missed.

Definition 5.4 (Disable Set). *The set of transitions that can disable a marking and level satisfied transition $t \in T$ given the set of transitions $T_\nu \subseteq T$ that fired to* *create the current Boolean state of the system is the union over the following sets:*

1. ***Marking Disabling**: $\{t' \in T \mid \bullet t \cap \bullet t' \neq \emptyset\}$; and*

2. ***Level Disabling**: $\mathsf{min\_set}(\bigcup_{T' \in \mathsf{unate\_solver}(\mathcal{T}_\mathsf{n}, T_\nu)} \mathsf{opposite}(T')) \cup \{t' \in T \mid \exists T' \in \mathsf{lrs}(t') : t \in \mathsf{opposite}(T')\}$, where the function $\mathsf{min\_set}(\mathcal{T})$ returns $T' \in \mathcal{T}$ such that for all $T'' \in \mathcal{T}$, $|T'| \leq |T''|$, $\mathcal{T}_\mathsf{n} = \mathsf{keep}(T_\nu)(\mathsf{lrs}(t))$ and $\mathsf{opposite}(T') = \{t'' \in T \mid \exists t' \in T' : (L(t') = w+ \wedge L(t'') = w-) \vee (L(t') = w- \wedge L(t'') = w+)\}$*

the function $\mathsf{disable}(t, T_\nu)$ returns the resulting set from the union.

The intuitive understanding of the disable set follows the two examples in Fig. 5.1. The marking disabling relates to Fig. 5.1(a); the level disabling relates to Fig. 5.1(b). The marking disabling is symmetric due to the structure of the net. The relation includes what t can disable, as well as what it disables. The level disabling is made symmetric in a similar fashion. The first set includes any transition that can disable t. This is the smallest set of transitions that can disable a product term in the Boolean function for t. It is computed from the causal group sets using only transitions that have fired to create the current Boolean state. The smallest set of these disabling transitions gives the best performance in the partial order reduction. The second set, however, is any transition that can be disabled by the firing of t. This gives symmetry similar to the marking disabling set. Unlike the first direction, however, this direction is not state dependent. All transitions that can potentially be disabled by the firing of t must be included in the level disabling set. Consider the example in Fig. 5.1(b). If t is equal to t_1, then the second part of the level disabling set includes t_2 since firing t_1 disables t_2. If t is equal to t_2, however, then the first part of level disabling includes t_1 since again the firing of t_2 disables it. This symmetric relation is critical to the correctness of the partial order reduction.

This definition of dependence is not sufficient to divide a set of fireable transitions into dependent and independent sets because it does not consider all of the independence conditions from Definition 5.3. There are additional transitions for which firing orders must be explored to be able to validate the level-ruled Petri

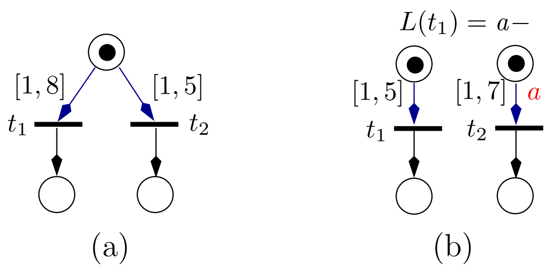


Fig. 5.1. Two level-ruled Petri nets that contain transitions that can be disabled. (a) A level-ruled Petri net that shows a disabling through a change of marking. (b) A level-ruled Petri net that shows a disabling through a change in Boolean state.

net correctness properties in the target submodule through a reduced state space. There are three additional ways that a failure can occur in a module that are not covered by the disable set: safe, consistent state assigned, and missed causality. The output semimodular and constraint properties are not included in this list because they are covered by either the disable set or the failure check as described next.

The partial order reduction explores marking and level disabling by default if they exist in the flat state space of the system. This covers the output semimodular property. A constraint violation occurs when a transition fires and it has a constraint rule that is not marking, level, or time satisfied at its firing. The violation must be found by the partial order if the constraint rule belongs to the target module. This failure directly relates to the notion of hide-fail in [56]. Consider the level-ruled Petri net in Fig. 5.2(a). Transition t_3 is in the target submodule. If transitions t_1 and t_4 fire, then the constraint rule r_1 is marking and level satisfied. The constraint rule may or may not violate its timing bounds depending on the amount of time that can elapse before t_3 fires. If transition t_3 fires before t_1 or t_4, however, then there is a constraint failure. Yoneda explores the firing orders of $\{t_1, t_3\}$ and $\{t_4, t_3\}$ if they exist in the full state space of the system [56]. These orders do not need to be explored in this implementation due to the way in which timing failures are detected. Timing failures are checked by looking at the separation between when a constraint rule is activated and a recently fired transition. If this separation is beyond the latest firing time of the constraint rule, then there is a latest firing time failure. Although a single firing sequence is explored in the reduction, the zone implicitly represents all possible firing sequences of its transitions; thus, the failure is detected regardless of the explored firing order.

Checking timing failures in this manner does have a drawbacks. If a system never fires any transitions, then timing failures cannot be checked because a fired reference transition is required for the check. Also note that the latest firing time failure check is conservative in that it can produce a false negative result. This is due to the partial order in the timing information method itself, as it represents multiple firing sequences in the same zone. The timing failure check can be made exact by individually looking at each firing sequence allowed by the zone. Orders are added to the zone to reflect the current sequence being checked. This correctly adjusts the separations according to the actual firing order and false negatives are thus avoided. This check, however, is complex and costly; thus, the check is performed on the partially ordered zone that contains the largest separations found in any firing order of its transitions.

A safety violation occurs whenever a place that already exists in the marking is added again by the firing of a transition. This violation must be found by the reduction if the place belongs to the target submodule, and if the violation exists in the complete state space of the system. Note that all violations are failures, only it is acceptable for the partial order reduction to not find failures that do not belong to the target submodule as they will be found in a later verification. Consider the level-ruled Petri net for a module shown in Fig. 5.2(b). The firing of t_1 can add p_1 to the marking for a second time. If t_2 fires first, however, then there is no violation. If p_1 belongs to the target submodule, then the partial order reduction must explore both firing orders of t_1 and t_2 if both orders exist in the full state space; otherwise, safety failures in the target submodule are missed. Although t_1 is not enabled in the drawn marking, it can become enabled by firing t_0. The partial order reduction must consider this.

A consistent state assignment violation occurs whenever a transition on a signal fires and does not toggle the state of the signal because the signal is already in the

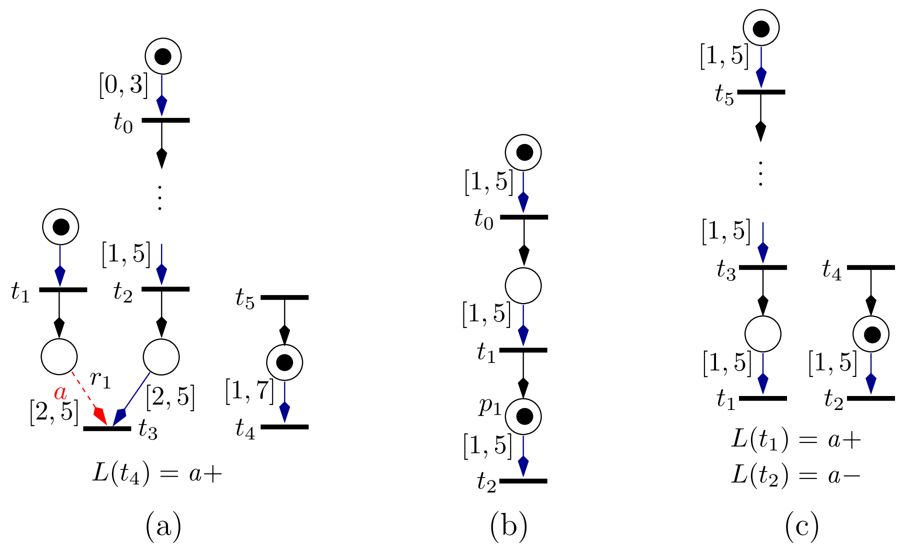


Fig. 5.2. Level-ruled Petri nets that show order dependent failures. (a) A level-ruled Petri net that may have a constraint rule failure. (b) A level-ruled Petri net that may have a safety failure for p_1. (c) A level-ruled Petri net that may have a consistent state assignment failure on signal a.

correct final state. This failure must be found by the reduction if the fired transition belongs to the target submodule. Consider the level-ruled Petri net in Fig. 5.2(c). The signal a is an output in the target module. Transition t_1 is labeled as $a+$ while t_2 is labeled as $a-$. Assume that the signal a is currently low. If transition t_1 fires followed by t_2, then there is no failure. If the other order is explored, however, then the module fails. Although t_1 is not enabled in the drawn marking, both orders of t_1 and t_2 must be explored in the partial order reduction if they exist in the full state space of the system; otherwise, consistent state assignment failures in the target module are missed. This means that the reduction may need to consider firing t_5 to see if t_1 can ever become enabled concurrently with t_2. The partial order must, in fact, explore all allowed firing orders of transitions defined on output signals of the target submodule for modular synthesis.

A missed causality can occur in a transition with a disjunctive Boolean function. Consider a transition t with a single rule that has the expression $a \vee b$. The signal a is high in the current Boolean state while by b is low. The rule is marked. There are only two causal assignment in this state: the transition that marked the rule and the transition that fired to make the signal a high. The partial order reduction must consider the possibility of being causal on the signal b too. If it is possible to fire a $b+$ transition before t is forced to fire on any of its other two causal assignments, then the reduction must consider this causality. If it does not, then it can miss a timing behavior on t that can lead to a failure because t may be able to fire earlier or later in time on the b causality.

The fail set represents additional transitions to the disable set whose firing orders must be explored by the partial order reduction if the orders exist in the flat state space of the system.

Definition 5.5 (Fail Set). *The fail set for a transition $t \in T$ given a module M_i in a larger system $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is the union over the following sets:*

1. ***Safe**: $\{t' \in T \mid \exists p \in P_i : p_i \in (t'\bullet \cap \bullet t)\}$;*
2. ***Consistent State Assigned**: $\{t' \in T \mid \exists w \in O_i : (L(t) = w+ \vee L(t) = w-) \wedge (L(t') = w+ \vee L(t') = w-)\}$; and*
3. ***Missed Causality**: $\mathsf{min\_set}(\mathsf{required\_set}(t, T - T_\mu, T - T_\nu))$, where $\mathsf{min\_set}(\mathcal{T})$ returns $T' \in \mathcal{T}$ such that for all $T'' \in \mathcal{T}$, $|T'| \leq |T''|$, and T_μ and T_ν are the fired marking and Boolean state transition sets;*

the function $\mathsf{fail}(t)$ returns the resulting set from the union.

The fail set completes the conditions of independence in Definition 5.3 that are not covered by the disable set. It contains a contribution for each of the failure conditions that are not covered by disable in Definition 5.4 as shown by the level-ruled Petri nets in Fig. 5.2.

The visible set for a next-time free linear temporal logic is given by Minea in [25]. A transition is visible if it appears in any formula describing properties of the system. This includes time constraints and signal valuations. The partial order reduction must explore all firing orders of visible transitions that exist in the flat state space of the system.

Partial order reduction traverses a portion of the reachable state space by not exploring all firing orders of independent and invisible transitions. It must explore, however, all orders of dependent and visible transitions that exist in the flat state space of the system.

Definition 5.6 (Relevant Set). *The relevant transitions to a fireable transition $t \in T$ is $\mathsf{relevant}(t) = \{t\} \cup \mathsf{disable}(t) \cup \mathsf{fail}(t) \cup X$, where X is the empty set if t is invisible; otherwise it is the set of visible transitions.*

The partial order reduction must explore all firing orders of transitions in $\mathsf{relevant}(t)$ if they exist in the reachable state space of the system.

The partial order reduction must determine which orders of relevant transitions exist in the complete state space. Consider the level-ruled Petri net in Fig. 5.1(a). Transitions t_1 and t_2 are fireable from the drawn marking regardless of the Boolean state or the valuation of timers on their rules. The relevant set for t_1 includes t_2 and both firing orders must be explored. Consider now the level-ruled Petri net Fig. 5.2(c). If the current timed state class s of the system is such that the fireable set is $T_f(s) = \{t_5, t_2\}$, then the partial order reduction must choose the ample set $\mathsf{ample}(s) \subseteq T_f(s)$ such that every transition not in $\mathsf{ample}(s)$ is not relevant to transitions in $\mathsf{ample}(s)$. The relevant set for t_2 is the set $\{t_1, t_2\}$. Transition t_1, however, is not in the fireable set $T_f(s)$. The partial order reduction must determine if there exists in the complete state space a state where t_1 is fireable with t_2; thus, it must determine in this example if the firing of t_5 can lead to a state s' such that $T_f(s') \supseteq \{t_1, t_2\}$.

The backward search of the level-ruled Petri net to find fireable transitions that can enable relevant transitions is exactly the necessary set algorithm presented in Section 4.6.2 of Chapter 4. The initial visited set is computed by Definition 4.39, only t_p is now t where $t' \in \mathsf{relevant}(t)$ and the necessary set for t' is being computed. Recall that the return type from the algorithm is a set of transition-delay pairs. Consider the level-ruled Petri net in Fig. 5.3(a). Transitions t_0 and t_1 are fireable in the current timed state class s. The relevant set for t_1 is $\{t_1, t_3\}$. Although t_1

is fireable, t_3 is not. The necessary set for t_3 is $\{(t_2, 3)\}$. A possible ample set for this state is $\mathsf{ample}(s) = \{t_1, t_2\}$, because the latest time after t_5 that t_0 fires is under 4, its takes at least 4 time units before t_4 fires. Transition t_1 does not have to fire for 8 time units; thus, there may exist a state after firing t_2 where t_1 and t_3 are concurrently fireable. A slightly more complex example is shown in Fig. 5.3(b). The state s is such that $T_f(s) = \{t_3, t_8\}$ and the signal a is high and the signal c is low. The relevant set for t_8 is the set $\{t_7, t_8\}$ because t_7 can level disable t_8. Although t_8 is fireable, t_7 is not. The necessary set of t_7 in this example is $\{(t_3, 3)\}$; thus, a possible ample set for the state is $\{t_3, t_8\}$ because again, assuming a suitably small separation exists in the zone of the state, there may be enough time to fire t_3, t_6, and then t_4 such that t_7 and t_8 can be concurrently enabled before t_8 is forced to fire.

Partial order reduction only needs to consider firing orders on relevant transitions if they are actually allowed by the level-ruled Petri net model in the reachable state space. Consider again the example from Fig. 5.3(a). A possible ample set for this example is $\mathsf{ample}(s) = \{t_1, t_2\}$ because t_3 is relevant to t_1; it is not fireable;

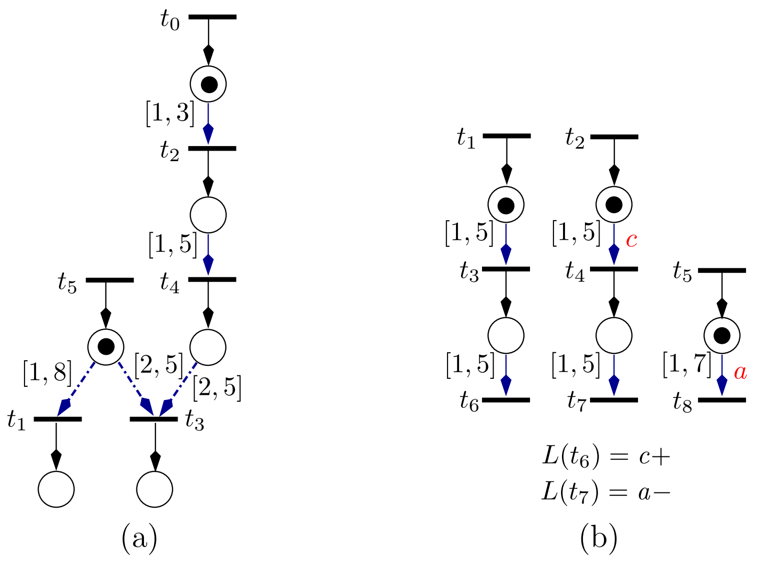


Fig. 5.3. Two level-ruled Petri nets with conflicting transitions that are not fireable together. (a) A level-ruled Petri net fragment with conflicting transitions t_1 and t_3. (b) A level-ruled Petri net fragment with conflicting transitions t and t_1.

and the necessary set for t_3 is $\{(t_2, 3)\}$. The delay from the necessary set can be used to determine if firing t_2 from this state actually leads to a new state s' where $T_f(s') = \{t_1, t_3\}$.

Definition 5.7 (Active Transitions). *A transition-delay pair (t_n, d) is active for a transition $t_d \in T$ given the zone z containing transitions $T_z \subseteq T$ if there exists ${t_d}' \in \mathsf{needs}(t_d) \cap T_z$ and ${t_n}' \in \mathsf{needs}(t_n) \cap T_z$ such that $\delta_L + \mathsf{Lft}({t_d}', t_d) \geq d + \mathsf{Lft}({t_n}', t_n)$; where δ_L is the min-max entry $({t_d}', {t_n}')$ in z; $\mathsf{active}(t_d, t_n, d, z, T_z)$ is true if (t_n, d) is active for t_d given z and T_z, and it is false otherwise.*

The pair (t_n, d) is active for t_d if t_d cannot fire later than t_n plus the delay d in the zone. The function $\mathsf{active}(t_1, t_2, 3, z, T_z)$ returns true if the min-max entry on (t_5, t_0) is greater than -3. If it is less than -3, however, then the function returns false because t_1 always fires before t_3 can become fireable; thus a possible ample set for this example is now $\mathsf{ample}(s) = \{t_1\}$, which yields a reduction since $\mathsf{ample}(s) \subset T_f(s)$.

The discussion this far has not considered the effect of replacing nonfireable transitions in the relevant set with active fireable transitions from the necessary set. Property 5.2 requires that transitions not in $\mathsf{ample}(s)$ be independent of any transitions in $\mathsf{ample}(s)$ for the reduction to be safe. If a necessary set transition replaces a relevant set transition, then any transition relevant to the newly added transition must be included also.

Definition 5.8 (Dependent Set). *The dependent set of a transition $t \in T$ that is marking and level satisfied by the current timed state class $s = (\mu, \nu, z)$ given the fired three-tuple $\mathsf{FT} = (T_\mu, T_\nu, T_z)$ is the fixed point $\mathsf{dependent}(T_d, s, \mathsf{FT}) = T_d$ created from the initial seed $T_d = \{t\}$, where $\mathsf{dependent}(T_d, s, \mathsf{FT})$ is the set of transitions $t' \in T$ such that there exists $t_d \in T_d$ where $t' \in \mathsf{relevant}(t_d)$ and $(\mu, \nu) \vdash R(t')$; or $t'' \in \mathsf{relevant}(t_d)$, $(\mu, \nu) \nvdash R(t'')$ and there exists a delay $d \in \mathbb{Q}^+$ such that $(t', d) \in \mathsf{necessary}(s, t'', T_V, \mathsf{FT}, \mathsf{false})$ and $\mathsf{active}(t_d, t', d, z, T_z)$ is true, where T_V is set by Definition 4.39 for $t_p = t''$; $\mathsf{dependent}(\{t\}, s, \mathsf{FT})$ returns the fixed point for the initial seed $\{t\}$.*

The dependent set is a fixed point calculation on an initial seed t. Any transition t' that is relevant and fireable to t is added to the set, or any transition t' that is necessary for some transition t'' and is active to t is added to the set too. The transitions in the dependent set of the added transitions must also be incorporated into the dependent set of t too. The final fixed point of the dependent set is the set of transitions for which all firing orders must be explored by the reduction.

The best ample set is the smallest dependent set containing only fireable transitions at the current state.

Definition 5.9 (Ample Set). *An ample set for the timed state class s given the set of fireable transitions T_f from s and the fired three-tuple* FT *is* $\mathsf{ample}(T_f, s, \mathsf{FT}) = \mathsf{min\_set}(\{T' \in 2^T \mid T' = T_f \vee (\exists t \in T_f : T' = \mathsf{dependent}(\{t\}, s, \mathsf{FT}) \wedge T' \subseteq T_f)\})$*; where the function* $\mathsf{min\_set}(\mathcal{T})$ *returns $T_a \in \mathcal{T}$ such that for all $T'' \in \mathcal{T}$, $|T_a| \leq |T''|$.*

The algorithm in Section 4.5 from Chapter 4 to traverse the timed state space of the level-ruled Petri net implements the partial order reduction by only computing successor states from the ample set instead of the fireable set. Verification in the reduced state space is correct if the computation of the ample set satisfies the properties in Section 5.1.3 as proven in [25]. Property 5.1 is satisfied because the ample set is empty only if the fireable set is empty. Property 5.2 is satisfied by virtue of the dependent set. Property 5.3 is satisfied because visible transitions are always included in the relevant set of a transition if it is visible too. Property 5.4, finally, is satisfied by a simple restriction in the nets: no zero loops and no infinite delays on transitions are allowed in any loops. This forces time to always advance; thus, transitions with bounded latest firing times must eventually fire and cannot be infinitely ignored.

5.2 Modular Synthesis

The parallel composition of the network of modules, M, can generate a large reachable timed state space. Modular synthesis requires knowledge of the timed states of this larger timed state space that relate to module M_i. These states are represented in a *reduced state graph* for the module in the larger state space.

The reduced state graph for a target subcircuit is derived from the reachable timed state class sequences of the larger system. The reduced state graph does not include timing information, and it only keeps information related to signals and transitions that are visible to the target subcircuit.

Definition 5.10 (Module Transitions). *The set of transitions that are visible to the module M_i in a larger system $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is $\mathcal{V}(M_i) = \{t \in T \mid t \in T_i \vee \exists w \in W_i : L(t) = w+ \vee L(t) = w-\}$, where T_i and W_i are the transition and signal set for the module M_i.*

The module transition are all transitions observable to the target submodule. These are transitions that affect its marking or the state of signals on its interface. The set is used to build the reduced state graph. All transitions in the set are visible in the partial order reduction.

Definition 5.11 (Reduced State Graph). *The reduced state graph $\mathsf{rsg}(M_i)(\mathcal{F})$ for a module M_i in a network of modules $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$ given a set of sequences $\mathcal{F}$ for M is $\mathsf{rsg}(M_i)(\mathcal{F}) = \{(\mathsf{proj}(s, M_i), t, \mathsf{proj}(s', M_i)) \mid (s, t, s') \in \mathcal{F} \wedge t \in \mathcal{V}(M_i)\}$, where a timed state class is $s = (\mu, \nu, z)$, W_i is the signal set in M_i, and $\mathsf{proj}(s, M_i) = (\nu \cap W_i)$.*

Consider the system in Fig. 5.4(a). This is a standard *OR* gate with its output passed through invert gates and used as input. The level-ruled Petri net modules for each of the components in the system are shown in Fig. 5.4(b), Fig. 5.4(c), and Fig. 5.4(d). The graphical representation of $\mathcal{F}(M)$ for the system in Fig. 5.4(a) is shown in Fig. 5.5(a). It does not include any timing information. If M_i is selected to be the level-ruled Petri net for the gate implementing signal a shown in Fig. 5.4(b), then the graphical representation of $\mathsf{rsg}(M_i)(\mathcal{F}(M))$ is shown in Fig. 5.5(b). The reduced state graph only contains information that is visible to a gate implementing the signal a and does not include any timing information. This is the input to the synthesis algorithm.

The reduced state space for a module in a system is created by Definition 5.11.

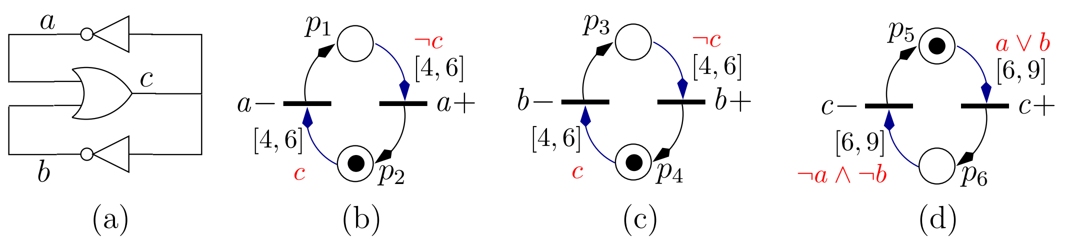


Fig. 5.4. A simple timed circuit with its level-ruled Petri net model (a) The gate schematics for the timed system. (b) The level-ruled Petri net of the *OR* gate producing the signal c. (c) The level-ruled Petri net of the invert gate producing the signal a. (d) The level-ruled Petri net of the invert gate producing the signal b.

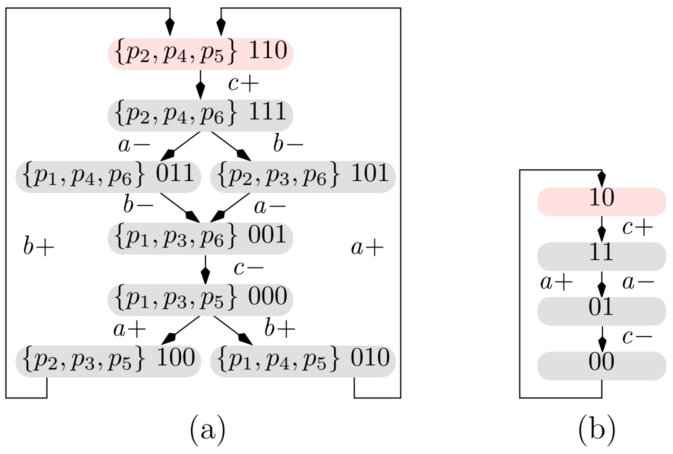


Fig. 5.5. The graph of the edge set and its reduced form. (a) The complete graph. (b) The reduced form of the graph.

The definition, however, requires the entire set, $\mathcal{F}(M)$, of reachable timed state class sequences for the larger system to first be generated since the reduced state space is built from this set. A modular approach using this method has no advantage because it must traverse the entire reachable timed state space of the system. It then discards many of the states found in the traversal to create the reduced state graph for the target submodule. The goal of this research is to avoid enumerating the full timed state space representation; thus, fewer states are discarded in generating the reduced state graph of the target submodule. This work defines a set $\mathcal{F}_{M_i}(M)$ that is a subset of $\mathcal{F}(M)$ but contains all information necessary to show correctness in the target subcircuit and to construct a reduced state graph for it. The assumption is that the cost of generating the reduced set of timed state class sequences, $\mathcal{F}_{M_i}(M)$, is small enough that it is more efficient to analyze each subcircuit individually than to consider the entire system at once.

The correctness of the proposed reduction is shown by looking at explored firing orders in $\mathcal{F}_{M_i}(M)$. This set must be related to $\mathcal{F}(M)$, and it must be shown to contain the same firing orders of transitions and signals visible to M_i as does $\mathcal{F}(M)$. If this does not hold, the reduced state graphs from the two sets are not equal; thus, synthesis could produce an incorrect circuit from the reduced set.

Definition 5.12 (Untimed Project). *The untimed project function removes from a timed state class sequence* $(\mathbf{s}, \mathbf{t})$ *entries that are not visible to a target module* M_i *in a larger system* $M = \mathcal{E} \parallel M_1 \parallel M_2 \parallel \cdots \parallel M_n$*:*

$$\mathsf{project}(M_i)(\mathbf{s}, \mathbf{t}) = \begin{cases} (\epsilon, \epsilon) & \textit{if } \mathbf{s} = \epsilon \vee \mathbf{t} = \epsilon; \\ ((\mathsf{proj}(s_0), X), (t_1, Y)) & \textit{if } t \in \mathcal{V}(M_i); \\ (X, Y) & \textit{otherwise}; \end{cases}$$

where $(X, Y) = \mathsf{project}(M_i)((s_1, \ldots, s_n), (t_2, \ldots, t_n))$*,* $\mathbf{s} = (s_0, s_1, \ldots, s_n)$*, and* $\mathbf{t} = (t_1, t_2, \ldots, t_n)$*; the function* $\mathsf{project}(M_i)(\mathcal{F}) = \bigcup_{(\mathbf{s},\mathbf{t}) \in \mathcal{F}} \mathsf{project}(M_i)(\mathbf{s}, \mathbf{t})$ *is the set of untimed projected sequences in* $\mathcal{F}$*.*

The untimed project function relates the two sets $\mathcal{F}_{M_i}(M)$ and $\mathcal{F}(M)$. It removes from a set of timed state class sequences any transitions that are not visible to the target submodule. The primary difference between Definition 5.12 for untimed project and Definition 5.11 for the reduced state graph is that Definition 5.12 preserves the firing sequences of the target submodule where Definition 5.11 does not. Definition 5.12 returns a set of timed state class sequences whereas Definition 5.11 returns a relation showing connectivity between timed state classes. A sequence and a connectivity relation are different in that sequences can be derived from the connectivity relation that do not exist in the set of sequences from Definition 5.12. This is an important subtlety in the proof of modular analysis.

The reduced set of timed state class sequences, $\mathcal{F}_{M_i}(M)$, must be defined to capture all of the behaviors of the target module M_i. Modular analysis of a target submodule, M_i, in a system of modules, M, is exact if $\mathcal{F}_{M_i}(M)$ is constructed such that the following property holds.

Property 5.5. $\mathsf{project}(M_i)(\mathcal{F}_{M_i}(M)) = \mathsf{project}(M_i)(\mathcal{F}(M))$

The property assumes that $\mathcal{F}(M)$ is exact. If Property 5.5 holds, then the following theorem is proven to show that the proposed reduction yields an exact reduced state space for the synthesis algorithm.

Theorem 5.1. $\mathsf{rsg}(M_i)(\mathcal{F}_{M_i}(M)) = \mathsf{rsg}(M_i)(\mathcal{F}(M))$

Proof. The proof shows that the existence of an edge in the reduced state graph derived from the complete reachable set forces the existence of the same edge in the reduced state graph derived from the reduced reachable set. The other direction is proven similarly but not shown. Suppose that there exists a timed state class sequence $(\mathbf{s}, \mathbf{t})$ in $\mathcal{F}(M)$ and an index $j \in \mathbb{N}$ such that $1 \leq j \leq [\mathbf{t}]$, t_i is a visible transition in $\mathcal{V}(M_i)$, and $(\mathsf{proj}(s_{j-1}, M_i), t_j, \mathsf{proj}(s_j, M_i))$ is found in the reduced state graph $\mathsf{rsg}(M_i)(\mathcal{F}(M))$. If Property 5.5 holds, then there must also exist a timed state class sequence $(\mathbf{s}', \mathbf{t}')$ in $\mathcal{F}_{M_i}(M)$ such that $\mathsf{project}(M_i)(\mathcal{F}) = \mathsf{project}(M_i)(\mathcal{F}')$; thus, there exists in index $k \in \mathbb{N}$ such that $1 \leq k \leq [\mathbf{t}']$, $t_k = t_j$, and $(\mathsf{proj}(s_{k-1}, M_i), t_k, \mathsf{proj}(s_k, M_i))$ is in the reduced state graph $\mathsf{rsg}(M_i)(\mathcal{F}_{M_i}(M))$ too. $\square$

The task is to construct $\mathcal{F}_{M_i}(M)$ such that Property 5.5 holds. This is readily accomplished by making all transition in $\mathcal{V}(M_i)$ visible to the partial order reduction. This forces the reduction to explore every possible firing order of signals and transitions in the target submodule. The net effect is for the reduction to produce a complete timed state space for the target submodule and a reduced state space for all things outside of the scope of the target submodule; thus, Property 5.5 holds.

5.3 Related Work

Partial order reduction is an important tool in mitigating state explosion in verification [67, 68]. Partial order reduction is applied to synthesis in [69, 70]. The approach in [70] is an unfolding technique that is applied to untimed specifications. Not only is it not clear if the technique can be efficiently applied to a timed model, the technique ignores hierarchy in the specification; thus, it is limited in the size of systems it can be applied to. The approach in [69] exploits hierarchy in the specification by applying a partial order reduction to signals not on the interface of the target subcircuit. It modifies the partial order reduction method in [56, 67] to always include all allowed orders of signals on the interface and in the target subcircuit. It then uses the state space based synthesis approach in [42] to produce an exact circuit. The work demonstrates a significant reduction in running time for state space exploration in the synthesis problem and greatly increases the size of systems that can be analyzed. The approach, however, is tied to the time Petri net. This negatively impacts the size of the reduced state space due to the structural complexity of the model.

This research extends the modular synthesis approach in [69] to level-ruled Petri nets to further reduce the size of the reachable state space through syntactic abstraction. The partial order in the timing information in [10, 65] is similar to that found in [24, 25], and like the approach in [25] does not require extra reference clocks for synchronization. The basis for the new algorithm is actually presented in [25] and is based solely on the time separation of transitions, but an initial implementation on timed Petri nets in [71] shows it to be incorrect for Boolean expressions and incomplete for partial order reduction; thus, this work corrects the algorithm and completely derives the conditions necessary to preserve correctness in the reduction. The partial order reduction is restricted to safe nets and works for any type of choice structure. The partial order reduction uses untimed methods from [67] and timed methods in [24, 25, 56] to determine independence between transitions. It augments these definitions to incorporate the notion of independence in the presence of Boolean functions. The definitions are not only tied to the structure of the net, but also consider the timing of transitions.

Interface abstraction is a common approach to reduce the cost of state based synthesis by exploiting hierarchy in the specification. As performing the abstraction by hand is error prone, work in [72] automates the abstraction process. It alters the actual system model by removing from it transitions that are not on the interface of the target subcircuit. Although the simplified model structure reduces the reachable state space, the approach is limited in the transitions it can remove, is not efficient on specifications with Boolean functions, and can produce nonexact circuits due to conservative timing behavior from the abstraction [72]. The work described in this paper does not alter the specification. It reduces the state space by exploring a single firing order on independent transitions not in the target subcircuit.

In [73, 74, 75, 76], partial cubes are used to conservatively explore reachable state spaces. A partial cube denotes that a state consists of primary inputs and outputs coupled with a boolean cube representing values for internal signals. In any given state, an internal signal may be known or unknown, so more states are included than may actually be reachable by the circuit. In abstracting out internal signal behavior in the state space, partial cubes can potentially achieve an exponential reduction in the size of the state space.

5.4 Summary

This chapter describes a partial order reduction for the level-ruled Petri net to verify the correctness properties from Chapter 3 in a reduced reachable state space of the system. The partial order reduction follows published work in the area. It adds to the published work an exact definition of relevant transitions used in the partial order reduction.

This chapter describes a modular synthesis approach that relies on the partial order reduction. The approach expands the set of visible transitions in the partial order reduction to include all transitions in the target module, as well as all transitions on signals that are visible to the target module. The partial order reduction is then able to produce the exact state space of the target module using the new visible set without exploring all the reachable states of the larger system. A circuit is synthesized from the reachable state space of the module using an algorithm in [42]. The circuit is exact when the timing analysis algorithm is exact.

CHAPTER 6

RESULTS AND ANALYSIS

The goal of this chapter is two pronged: first, to evaluate the performance of the timing analysis algorithm; and second, to evaluate the impact of the modular approach. Neither of these goals is easily addressed. Generalities are fraught with issues because there are so many variables that affect performance. It is especially difficult to analyze the performance of algorithms that are operating on a new model—the level-ruled Petri net. It is almost necessary to adapt and implement prior algorithms to the new model. This defeats the purpose, however, of the evaluation because the adapted algorithms now become new algorithms running on a new model. There is no obvious unbiased means to overcome this obstacle.

Section 6.1 is an analysis of the timing analysis algorithm presented in this work. Section 6.2 is an analysis of the modular synthesis reduction technique. The two sections open with an explanation of the analysis approach. Both sections show favorable and less than favorable results for the work in this dissertation. This is important in correctly setting the potential for this work. It is important to note too that several of the benchmarks for the analysis come from industrial designs. These include the analysis of circuits from the Intel RAPPID decoder [1], the IBM gigahertz processor in [4], the STARI FIFO (recently appearing in a product from Sun Microsystems [14, 15]), and the gasp pipeline controller again from Sun Microsystems [2].

This chapter concludes with an industrial scale design from IBM. Section 6.3 analyzes and synthesizes circuits for IBM's synchronous interlocked CMOS pipelines [12]. It verifies general pipeline structures and suggests alternative circuits to published results that may yield better performance. This example is significant because it is beyond the capability of previous algorithms. This chapter concludes with a brief summary in Section 6.4.

All of the results for this chapter are run on a common machine. The machine, donated by Intel Corporation, is a Pentium III 930 MHz with 256Mb of memory. Although it is a competent machine, it is not a compute server. The results and analysis are to be understood with this in mind. The new timing analysis algorithm is implemented in the CAD tool `ATACS`. `ATACS` is a tool for the synthesis and verification of timed circuits [42, 77]. The POSET timing algorithm, [43], is also implemented in this tool too making it a convenient algorithm to compare against.

Many of the designs in this chapter are based on published industrial pipeline or FIFO circuits. This suggests that perhaps the evaluation is incomplete because it does not consider a more heterogeneous system. Although this is a point of interest, it does not devalue the import of demonstrating timed circuit analysis applied to real design—even pipeline design. The pipeline stage is the basic building block in many digital designs. It is replicated across an entire chip, and it is one of the first places that designers are exploring in their quest to save power and reduce noise; thus, the development of tools and methodology to improve the analysis of pipeline stages seems important to industry. In addition, it demonstrates a class of circuits that commercial tools do not adequately address, and for which this type of analysis is well suited. These case studies help to crystallize the challenge of synthesis and verification in emerging circuit technologies.

6.1 Timing Analysis

The first analysis is a direct comparison to Belluomini's POSET timing method in [43]. The POSET timing algorithm is implemented for the timed event/level structure that closely resembles the level-ruled Petri net. The reader is referred to Section 2.4 for a discussion on the differences between the two system models. The comparison is appropriate because it is previous work from this group, and it sets the baseline performance metric for derivative algorithms on these types of system models. More specifically, it is the first known timing analysis algorithm to support syntactic abstraction in an event based model. The comparison benefits from the fact that the two system models can be restricted to a set of examples where their modeling power has no effect. This makes the comparison as close as possible to two different algorithms running on the same timed circuit model.

POSET timing analysis stores rule timer separation values in the zone. This is different from the new timing analysis algorithm in this work that stores transition separations in the zone. A close look at the POSET algorithm reveals that it uses two zones in building the finite representation: one containing rule separations and one containing transition separations. The new algorithm uses a single zone with transition separations. The comparisons in this section look at the number of zones used in the finite representation. The zones, however, contain very different information.

The first comparison to the POSET timing algorithm is for a regression suite of examples. The suite contains 122 examples. Most of the examples are for asynchronous circuits that have appeared in various publications over the last several years. All of the examples have running times that are under one second for both the new timing analysis algorithm and the POSET timing analysis algorithm. The amount of memory used to by each algorithm is near identical too because the examples are relatively small. The goal of looking at all of these examples is to show functionality in the new analysis algorithm.

The results of the comparison are shown in Table 6.1 and Table 6.2. The division between the two tables is arbitrary. It exists to accommodate the large number of examples in the regression suite. The column labeled *States* in the table refers to the number of unique marking and Boolean state pairs, (μ, ν), in the example. The columns labeled *Zones* is the number of zones needed to capture the complete timed state space of the example. This represents the cost of the timing information. The results from the new timing analysis algorithm are indicated with the *BAP* label (bourne again POSET timing analysis). The results from the POSET timing analysis are indicated with the *POSET* label.

TABLE 6.1
Comparison with POSET Timing on Regression Suite I

Example	States	BAP Zones	POSET Zones
abstract	6	6	6
alloc-ob	21	21	21
alloc-obusc	21	21	21
box1	25	25	31
box	25	25	34
case1	8	8	8
case2	10	10	10
case3	11	11	11
case4	11	11	11
case7	28	28	70
celement	8	8	10
choice2	51	51	102
circ1	8	8	16
circ2	6	6	8
circ3	8	8	14
circ4	9	9	20
c	6	6	6
cnt11	108	108	172
cnt3	32	32	48
cnt3_synch	64	64	89
coverlap	24	24	28
cstat	8	8	12
dlatch	10	10	10
DME	28	28	28
elatchB	38	39	69
elatch	37	38	67
etlatch2	30	36	63
etlatch3	55	60	104
etlatchP	60	63	113
ex1b	12	12	12
ex1	9	9	9
FIFO	16	17	27
FIFOR	14	15	23
FIFOSV	24	25	45
go	16	19	78
ifreq1	20	20	32
ifreq2	15	17	36
inv	4	4	4
jordi1P	20	20	41
jordi2	6	6	25
JSPelatch	72	73	180
lapb2	82	102	202
lapbN	82	102	202
lapb_pa	12	14	18
lapbsv	20	20	31
lecture7	20	20	35
loop	16	16	16
merge	33	33	58
mul2c	50	59	169
must-share1	24	24	28
nondisj	16	16	16
nosing	14	14	16
overlap	16	16	18
pab_a1	10	11	12
pab_b2	11	11	14
pab_c3	12	13	18
pab_c4_2	32	83	120
pab_c4	12	15	19
pab_c5	10	10	11
pab_c8	10	11	12
pab_c9	8	8	8
pif	9	10	11
pvuv	13	13	16
rcv-setup-usc	16	16	17
regions	5	5	7
rev	15	15	15
rlm	12	12	12
scsiP	17	17	26
scsiR	10	10	13
scsiSV2	22	22	42
scsiSV	16	16	23
scsiSVN2	50	136	220
SEL	58	66	180
selopt	64	64	362
silly	8	8	9
simple	6	6	6
simp	7	7	7
slatch2	25	26	34
slatch	30	31	48
sm	10	10	10
splitsemi	80	102	178
srdand	10	20	52
srdaoi	23	37	56
srdor	11	11	12
srgate	40	40	42
stariE	105	105	211

TABLE 6.2
COMPARISON WITH POSET TIMING ON REGRESSION SUITE II

Example	States	BAP Zones	POSET Zones
start6	50	50	61
statem	2	2	2
sunfifo2	29	30	60
tff	8	8	8
udding	66	66	91
upipe	11	12	13
var	12	12	15
varP	11	11	15
vmeP	14	14	18
x	8	8	11
xor	6	6	9
share	24	24	28
tag1	48	65	281
cgate	10	10	14
circ5	8	8	8
compare	53	53	56
hb	33	35	35
hw4	22	22	22
level	4	4	4
mul2	62	68	70
oneshot	3	3	3
pcfb	55	75	67
pchb	33	35	35
ring	4	4	4
scsiL	16	18	16
scsiSVimp	16	16	16
split	27	27	33
sunfifo	29	30	29
timedep	23	23	31
csa_tb	10	10	10
direct	4	4	4
killpack	71	73	71
spdor	47	47	47
tb_pre_dec	87	91	87
uno1_ctrl_tb	138	139	138
uno_ctrl_tb	138	139	138

A result that has equal values for the states and zones is ideal. The timing representation cannot be optimized beyond this point. There is a single zone for each unique (μ, ν) pair reachable by the level-ruled Petri net model of the example. The new algorithm meets this ratio for many of the examples in the regression suite. The new algorithm represents the timed state space using fewer zones than the POSET timing analysis in all but seven of the examples. These examples are found in Table 6.2: *pcfb*, *scsiL*, *sunfifo*, *killpack*, *tb_pre_dec*, *uno1_ctrl_tb*, and *uno_ctrl_tb*. The results for the *sunfifo*, *uno1_ctrl_tb*, and *uno_ctrl_tb* examples differ by one zone. The results for *pchb* differ by 8 zones. The results for *scsiL* and *killpack* differ by two zones. The results for *tb_pre_dec* differ by four zones. The average increase is 5% for these examples. The cause of the differences is a fundamental property of the new algorithm. Multiple causal assigned zones exist for a causal assignment. Each zone reflects an order on transitions in the assignment. If pruning can remove most of the transitions, then the number of causal assigned zones is reduced. If it cannot prune many transitions, however, then each causal assigned zone is unique and must exist in the final timed state space representation. The POSET timing analysis algorithm does not suffer from this because it abstracts away transition orders and looks only at the resulting rule orders. Two unique transition orders for a causal assignment can result in the same rule order in the POSET timing method; thus, it needs a single zone where the new algorithm may need two.

The new timing analysis algorithm outperforms the POSET algorithm in 81 examples—there are 33 examples where they post identical results. The new algorithm yields a 33% average reduction over the POSET method in the number of zones needed to represent the state space in these 81 examples. The improvement of the new algorithm over the POSET approach is most likely attributed to pruning in the zone. The new method is able to effectively prune in these examples; thus, reducing the number of zones at each untimed state. The new analysis algorithm gives a 22% average reduction over the POSET algorithm in the number of zones across all 122 examples in the regression suite.

Belluomini presents results for POSET timing on several examples in [43]. Table 6.3 is a summary of results from a selection of these examples. The selection is

TABLE 6.3
COMPARISON WITH POSET TIMING ON DISSERTATION EXAMPLES

	BAP				POSET			
Example	States	Zones	Mb	Time	States	Zones	Mb	Time
cnt7_synch	2048	2048	266	108.5	2048	4737	267	569.94
lapb4sv	885	1238	4	3.08	885	5321	8	2.01
lapb6sv	7930	15154	116	280.85	7930	160905	290	242.33
selector2	452	582	3	0.65	452	1553	5	0.68
selector3	5643	12962	28	61.01	5643	33569	108	132.17
tag7	2745	3361	13	13.09	2745	8641	65	26.14
tag_level	3949	6259	22	30.12	3949	4194	33	19.38
stari_old10	14529	19839	36	73.21	14529	14857	120	51.2
domino_compare	788	788	4	1.7	788	788	7	0.69
clz	8013	8021	36	40.8	7997	8024	77	60.01
mfxu	DNF	DNF	DNF	DNF	1549	6326	28	38.69

based on the complexity of an example, the application to this work, the level-ruled Petri net semantics and algorithm limitations, and the current state of `ATACS`. The trivial examples are omitted in the table because they are already shown in Table 6.1 and Table 6.2. These tables show that the new algorithm generally gives a better representation size for small examples. There are six other examples excluded from the table besides the trivial ones. Two of the examples generate an enormous number of zones to test the MTBDD zone representation. This is not implemented in this work because it results in a severe degradation in running time. Two are from the analysis of the IBM gigahertz processor in [13, 43]. These are the *pla* and *mle* circuits. The *pla* circuit uses nondisabling semantics. These are not supported by the level-ruled Petri net, so the new timing analysis method cannot be applied directly to the pla example. A modified version of the *pla* circuit that does not include nondisabling constructs completes in both POSET timing and the new algorithm giving identical results. The *mle* specification no longer completes in POSET timing analysis as implemented by the CAD tool `ATACS`.

The *alpha* and *beta* specifications are the two last examples that are not included in this analysis. The new timing analysis algorithm cannot be applied to these examples because the separation on the transitions is divergent. Consider the *beta* example in Fig. 6.1 with two transitions. The separation between successive $a+$ and $a-$ transitions is bounded in the 0 to 15 time unit range; and the separation between successive $a-$ and $a+$ transitions is bounded in the 0 to 15 time unit range. This is true for successive transitions on signal b, too. What about the separation, however, between $a+$ and $b+$? Is this separation bounded? No, it is not. Consider the case where $a+$ and $a-$ always fire 3 time units after being marking satisfied;

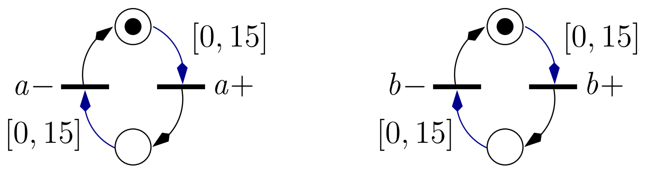


Fig. 6.1. The *beta* example with two signals a and b.

and $b+$ and $b-$ always fire 12 time units after becoming marking satisfied. The first instance of $a+$ is at time 3 and the first instance of $b+$ is at time 12. Their separation is 9. The second instance of $a+$ is at time 9, and the second instance of $b+$ is at time 36. Their separation is now 27. The separation between $a+$ and $b+$ increases with every instance of their firing; thus, the new timing analysis algorithm always grows this separation and never terminates exploration. The POSET timing algorithm does not suffer from this issue due to how it represents the timed state space with separations on rule timers rather than transitions. The new analysis algorithm can be applied only to examples that correlate transitions in some way. Fortunately, noncorrelated transitions seem to mainly exist in contrived examples that are designed to break analysis algorithms; at the least, this is the case with the examples analyzed in this work.

The comparison in Table 6.3 does not show one algorithm to be clearly superior to the other; although, memory is better managed by the new algorithm when compared to the old. The new algorithm completes in one-fifth the running time of the POSET algorithm for *cnt7_synch* and reduces the representation size by over half that of POSET algorithm. A similar improvement is seen for the *selector2* and *selector3* examples, as well as the *tag7* example. The new algorithm reduces the number of zones for the *lapb4sv* and *lapb6sv* examples over POSET timing, but its running time is above that of POSET timing. This increase in running time is most likely due to the pruning operation in the zone. Tracing backward in the level-ruled Petri net can be costly if the net is large. The *tag_level* example is a gate model of a tag circuit from the Intel RAPPID decoder with seven inputs that use syntactic abstraction [1]. The *tag7* example is a behavioral model of this same circuit that does not use syntactic abstraction. POSET timing produces better results for the *tag_level* circuit in both the number of zones and running time. This is also the case for the *stari_old10* example—a gate level model of 10 stages of the STARI FIFO in [16]. These results seem to suggest that the new analysis algorithm does not perform as well in highly concurrent specifications that use syntactic abstraction. The *domino_compare* example, however, uses syntactic abstraction, but the two analysis algorithms produce near identical results. The degraded running time in the new algorithm is a result of the cost of pruning the zones in the representation. It is more likely, that aside from the running time, the differences between the two representation sizes is a fundamental property of the algorithm.

The new analysis algorithm must split zones to create causal assignments where the POSET timing does not. The splitting usually occurs only in the presence of syntactic abstraction. Recall that a causal assigned zone given a causal transition is created for each member of the necessary set that contains the causal transition, as well as other transitions. If a transition has a necessary set of $\{\{t_1, t_2\}, \{t_1, t_3\}\}$, then the t_1 causal assignment creates two zones to order t_2 and t_3 with respect to t_1. This ordering is not reflected in the zone for the POSET timing algorithm because its zone contains rule separations instead of transition separations.

The final three examples in Table 6.3 are from the IBM gigahertz processor [13, 43]. The new timing analysis algorithm and the POSET timing algorithm do not agree on the number of unique marking and Boolean state pairs for the *clz* example. It is not known if this difference is a result of an implementation issue in either of the two algorithms. The example, unfortunately, does not complete for other analysis methods in `ATACS`, so the correct count is not known. The final example, *mfxu*, does not finish (DNF) in the new timing analysis algorithm. The number of zones generated by the new algorithm grows unbounded until system memory is exhausted.

Table 6.4 shows the results of pushing both algorithms to their very limits. The two algorithms are applied to increasingly larger designs until they are unable to complete. The first two entries in the table are 11 and 13 stage STARI FIFOs. These are behavioral specifications of the FIFO that do not use any syntactic abstraction. They are purely transition based. The new analysis algorithm outperforms the POSET timing algorithm in zones, memory, and time for both examples. The next two examples are again STARI FIFOs, only this time they are gate level descriptions of the FIFO that use syntactic abstraction for all of the gates. Note that this implementation is different from *stari_old_10* in the previous table.

TABLE 6.4
Comparison of Algorithms at Limits

	BAP				POSET			
Example	States	Zones	Mb	Time	States	Zones	Mb	Time
stari11	11508	13942	35	60.51	11508	56182	204	467.83
stari13	69756	88230	302	1093.49	DNF	DNF	DNF	DNF
lstari10	21428	36439	73	212.62	21428	21914	185	151.19
lstari12	41507	66205	155	542.67	DNF	DNF	DNF	DNF
gasp3	16288	28176	91	155.99	16288	20908	170	91.87

The new algorithm does not perform as well on this example. This is consistent with the analysis in Table 6.3. Also consistent with the earlier analysis is the better management of memory, since the new algorithm can complete twelve stages (*lstari12*) of the design without running out of memory. The final entry in the table is gate level description of the GasP FIFO by Sun Microsystems [2]—a minimal self-timed FIFO control. The use of syntactic abstraction degrades the performance of the new analysis algorithm as expected; though, the new algorithm uses significantly less memory.

The analysis seems to indicate that the new algorithm is better suited to designs that do not use syntactic abstraction. In the majority of the examples that fit this requirement, the new algorithm outperforms the POSET timing analysis in zones, memory, and running time. For the designs where syntactic abstraction is used, the new algorithm is competitive with the POSET timing. Although it usually shows a degraded zone count and running time, the differences are not too significant. More importantly, however, is that its superior memory management allows it to complete on larger designs. The author acknowledges, however, that this is more a result of implementation than that of algorithm design; although, the same argument can be made for running time too.

6.2 Reduction

This section first analyzes the impact of reduction by comparing it against flat analysis. The idea is to better appreciate what the modular analysis can do. Beyond this the analysis becomes complicated. The modular approach makes comparison to other methods complicated because there are not many that exists for timed systems outside of our research group. Those that do exist are geared to protocol, not circuit, verification; thus, they cannot be fairly compared to modular synthesis. This section compares modular synthesis to the two other methods that we have access too: automatic abstraction by Zheng in [72] and modular synthesis by Yoneda in [69]. Although these approaches are not independent of this work, they do serve as a reference for comparison.

Table 6.5 is a comparison between flat and modular synthesis. The goal is to see if the cumulative cost of having to analyze each component of a design in a reduced state space overpowers the savings in the modular approach. The table indicates that this is not the case for these three examples. The first three columns of the table show the cost of synthesizing each stage of the behavioral specification of the STARI FIFO. Each stage of the STARI FIFO is not identical in the design due to timing and initial state. The typical STARI FIFO can have up to three different cells from two literals to ten literals in size. The next three columns are the cost to verify each stage of a STARI FIFO circuit implementation. The final three columns

TABLE 6.5
Cost of Modular Analysis

	stari13			lstari12			gasp3		
	States	Zones	Time	States	Zones	Time	States	Zones	Time
1	459	752	1.7	542	679	1.52	396	478	2.43
2	463	723	1.61	544	681	1.52	529	553	4.68
3	467	701	1.78	544	677	1.53	616	972	14.88
4	655	1178	2.96	506	646	1.44			
5	758	1239	3.46	506	636	1.11			
6	658	1197	3.74	506	562	0.72			
7	1900	2426	8.47	675	979	2.23			
8	994	1345	5.05	604	880	2.13			
9	982	1070	4.3	506	562	0.75			
10	831	983	3.55	535	666	1.49			
11	984	1222	2.73	542	679	1.51			
12	439	584	0.94	544	680	1.49			
13	439	696	1.18						
	10029	14116	41.47	6554	8327	17.44	1541	2003	21.99

are the cost to verify each stage of a GasP circuit implementation. The number of zones and running time is trivial at each stage of all three designs. The total cost is shown in the last row of the table. These numbers show a dramatic improvement compared with those in Table 6.4 for the same specifications. If hand abstraction is used to create an environment for a single stage of any of these designs, then the analysis cost of the hand abstracted model of a single stage is around a hundred states and zones. This gives perspective to the effectiveness of the partial order reduction in these examples.

Table 6.6 tries to set new bounding sizes for the STARI FIFO and GasP examples. Although this is not reflective of every example, it does improve the number of stages that can be considered in these examples. The new analysis algorithm with the reduction can analyze the tenth stage in a STARI FIFO behavioral specification with 20 stages in under 72 minutes. Note, the analysis cost for all stages is not this large. The third stage is analyzed in under 5 minutes. These results do compare favorably to those in [27], where the same size STARI FIFO modeled by a timed automata is analyzed using discrete time semantics in over 10000 seconds using a Pentium II processor with 512Mb. Although it is important to temper the results by the fact that the analysis in [27] is a flat analysis, not a modular analysis. These results also show a dramatic improvement over those in [44], where hand abstraction is used. The gains for a stage (third stage) in the GasP example are not as dramatic due to the length of the running time, but they do push the limits out further. More importantly, however, is that memory is effectively managed in the reduction. These are not the actual limit for these examples, but they serve to establish that the reduction extends the ability of the timing analysis algorithm.

TABLE 6.6
Limits for Modular Analysis

Example	States	Zones	Mb	Time
stari20	39109	99995	163	4276.58
gasp6	929	2027	83	1575.24

Table 6.7 is a comparison to two other modular analysis approaches. The STARI FIFO is used for the comparison because the other two methods can only be applied to examples that do not use syntactic abstraction. Zheng presents an abstraction method in [72]. The approach is conservative and changes the physical structure of the level-ruled Petri net by combining rules and timing information on abstracted events; thus, the resulting circuit from synthesis may be larger than it needs to be. The first section of the table presents the abstraction method applied to the STARI FIFO with thirteen stages. The abstraction alone yields a significant reduction in the analysis of the STARI FIFO. It is not, however, as effective as the modular approach in this work as evidenced by Table 6.5. The two methods can be combined to produce a more significant reduction in analysis cost. The result of using both abstraction and modular synthesis is shown in the second section of the table labeled *Both*. This is a significant reduction in the analysis cost for each stage. The progressive nature of the states and zones is due to the amount of abstraction that can be performed given each stage of the design. There is more abstraction in the early stages than in the latter stages. The resulting circuit using abstraction may not be the exact minimal circuit in general. This is not, however, the case for the

TABLE 6.7
COMPARISON TO ABSTRACTION AND VINAS-P

	Abstraction			Both			VINAS-P		
	States	Zones	Time	States	Zones	Time	States	Zones	Time
1	1253	4164	17.12	51	80	0.07	611	942	0.24
2	617	870	2.69	62	73	0.1	549	880	0.21
3	854	1336	5.53	74	104	0.21	537	836	0.24
4	978	1601	6.99	86	92	0.11	641	1049	0.34
5	872	1757	5.25	104	135	0.3	648	1043	0.83
6	1068	1790	6.89	145	229	0.66u	1019	1535	1.83
7	808	1480	2.64	232	339	0.69	1379	1665	3.02
8	906	2033	5.47	338	630	1.44	1170	1337	1.11
9	996	1992	5.48	189	364	0.88	1033	1111	0.79
10	1800	2899	17.41	200	380	0.92	626	674	0.4
11	1804	2648	14.07	359	359	0.98	607	908	0.23
12	2484	3640	30.89	221	235	0.38	607	908	0.22
13	4168	6320	76.87	301	323	0.75	607	908	0.21
	18606	32530	197.3	2362	3333	7.49	10034	13796	9.67

STARI FIFO. Abstraction gives the exact results.

Yoneda presents a modular synthesis approach that is similar to that of this work in [69]. The primary difference is in the timing analysis algorithm and the system model. Yoneda does not implement a partial order in the timing information and uses the time Petri net for the circuit model. The results of the modular analysis for the STARI FIFO is shown in the third section of Table 6.7 labeled *VINAS-P*. These results are not generated on the same machine as the other results thus presented. These results are from a 1 gigahertz Pentium III with 2GB memory. They compare similarly to those in Table 6.5 from the new analysis algorithm. The running time for *VINAS-P*, however, is better than `ATACS`. This can be attributed to two things: first, a more efficient implementation; and second, the simpler semantics of the time Petri net. The level-ruled Petri net has a richer semantic structure due to the support for syntactic abstraction and the rule based timing model. This adds complexity to the analysis algorithm that does not exist in the time Petri net analysis algorithms. The total number of explored states and zones is near identical for the two analysis algorithms. The differences in the number of zones and states explored at each stage is due to different implementations of the partial order reduction. The two implementations more than likely explore different orders of independent transitions that lead to different levels of reduction.

6.3 Synchronous Interlocked Pipelines

An asynchronous pipeline is seemingly well suited to low power and noise sensitive applications because it eradicates the clock—the rightfully accused power abuser and noise generator in digital design. The lack of global synchronization on a clock, however, is not the source of the asynchronous benefit. It is the byproduct. A careful analysis reveals two properties that fuel low power and noise reduction in asynchronous pipelines: first, asynchronous pipeline stages only activate for valid data; and second, pipeline stages make local control decisions for data movement. Though this removes the need for a global clock, it more importantly reduces switching activity in the circuit; thus, energy consumption and noise generation is decreased. The remaining switching activity is also not correlated too. This leads to a further noise reduction. Asynchronous circuits do not generate spurious switching by design (i.e., designs are hazard or glitch free). The reduced switching activity results in power savings too. This is not, however, as significant a saving as that in selective activation and local control of data flow.

A typical synchronous pipeline stage is active on every clock cycle even if data is not present. A significant amount of power is wasted in the absence of data because almost all of the power is consumed at the leaf nodes of a clock tree. Synchronous pipeline stages rely on global control signals to dictate data flow. Local decisions on data movement cannot be made at the stage level. The asynchronous pipeline overcomes these issues by interlocking control signals in the forward and backward directions of the pipeline at the stage level of the design. A given stage cannot forward data to the next stage unless it has data to forward and the next stage is ready to accept that data. This forward and backward interlock in the pipeline implements the selective activations and local control of data flow.

The synchronous interlocked pipeline is a synchronous pipeline from IBM that interlocks control at the stage level in a manner similar to asynchronous pipelines [12]. The primary difference, however, is that the interlock is achieved using the global clock signal, along with control signals generated at the stage level of the pipeline. A traditional asynchronous pipeline uses only the stage level signals. A stage only activates on valid data; and a stage only forwards valid data when the downstream stage is ready to accept it. If a valid piece of data is to be forwarded to the next stage, then it is not sent until the clock edge, making the pipeline synchronously interlocked rather than asynchronously interlocked. A synchronous pipeline can now enjoy some of the benefits of asynchronous design while staying in the widely accepted and supported synchronous paradigm.

This section applies modular analysis to the synthesis and verification of IBM's synchronous interlocked pipeline structures. Many of these structures are beyond the capacity of the new analysis algorithm and can only be analyzed using the modular approach. Each section is devoted to a structure from [12]. It identifies where internal glitches exist in the design, and then suggests alternative circuit implementations found in modular synthesis that do not generate any internal glitches. Although the internal glitches do not affect the correctness of the initial implementation, they do contribute to noise generation and power consumption. The goal of this section is to show the application of modular analysis to real world design; and more importantly, on a design that can now be analyzed without needing to apply error prone hand abstractions.

6.3.1 Elastic Synchronous Pipelines

The elastic synchronous pipeline is key to the interlocked synchronous pipeline stage. The forward interlock signals valid data. This is readily implemented by a valid bit that moves forward in the pipeline with valid data. Stages activate only if the valid bit is set. The backward interlock is less obvious. The current stage signals to its upstream stage that it can accept data using the backward interlock signal. This implies that any stage in the pipeline can stall on any clock cycle if it cannot accept data; thus, it must be possible to progressively stall the synchronous pipeline at any point. Consider the scenario where the pipeline is completely full with valid data. Suppose that the last stage of the pipeline stalls and cannot accept new data. It asserts a control signal to its adjacent upstream stage indicating it is not ready for data. At the global clock edge, the upstream stage holds its data, but what happens with all of the other stages in the pipeline? They do not know anything about the stall at the end of the pipeline, so they forward their data; thus, data are lost at the stall boundary unless there exists extra storage to hold it until the stall is propagated to the front of the pipeline. This creates a simple formula: if there are N data items in a pipeline, then it takes $2N$ storage locations to progressively stall the pipeline assuming that data are inserted on each clock edge until the first stage is stalled.

The elastic synchronous pipeline implements the progressive stall and creates the backward interlock. It is shown in Fig. 6.2. The pipeline relies on two-phase clocking for correct operation. Data exist in every other stage of a two-phase

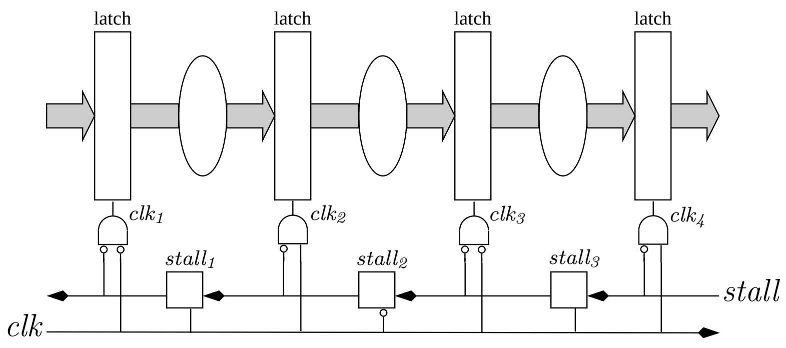


Fig. 6.2. Elastic synchronous pipeline.

clocked pipeline; thus, an N-stage pipeline holds $\frac{N}{2}$ data items when it is full. The *stall* signal is propagated backward in the pipeline by the one-bit latches. It moves backward one stage on each edge of the clock. The clock inputs to the data latches are gated by the respective stall signals at each stage. It takes N clock edges to propagate the *stall* signal from the last stage of the pipeline to the first stage of the pipeline; this is equivalent to $\frac{N}{2}$ clock cycles. If a new data item is inserted into the pipeline on every rising edge of the clock, then the pipeline holds N data items by the time it is stalled: $\frac{N}{2}$ data items are inserted into a pipeline that already holds $\frac{N}{2}$ data items before the first stage sees the stall signal. The extra $\frac{N}{2}$ data items inserted into the pipeline before it is stalled are inserted into the empty stages that naturally exist as a result of the two-phase clocking; thus, the empty stages become full stages when the pipeline stalls to absorb the latency of propagating the stall progressively backward in the pipeline.

A gate level model of the three stage elastic synchronous pipeline is analyzed by `ATACS`. The model uses syntactic abstraction to describe all gates, and it includes one-bit of the data path. The clock gating logic must not produce any glitches, and setup and hold times must be satisfied at the latches for the data and stall bits. This design is simple enough that it does not require modular analysis or synthesis. The new algorithm in `ATACS` verifies the correctness of the implementation in a few seconds. `ATACS` did, however, find an important timing assumption for correctness to hold. Data at a latch input cannot change faster than the delay through an inverter and a single *AND* gate. If this timing constraint is not met, then there is a race condition between the positive phase and negative phase adjacent pipeline stages. If there is no logic between the stages, then the noninverted inputs of the clock gating logic needs to be buffered to minimize clock skew in the design. This constraint, however, is typical in any latch based pipeline design.

6.3.2 Two-phase Interlocked Pipelines

The two-phase interlocked pipeline is shown in Fig. 6.3. The pipeline implements a forward and backward interlock using the *valid* and *stall* signals respectively with the *gclk* signal. A high *valid* signal indicates that there is valid data at the stage. A high *stall* signal indicates that the stage must hold its current data if it is valid and propagate the stall signal backward. If data at the current stage is invalid, however, then the *stall* signal is ignored at that stage and is not propagated backward. The stall signal is ignored until the stage has valid data.

The schematic in Fig. 6.3 is verified at a gate level for three stages by *ATACS*. If the data latches are omitted in the analysis, then the modular approach is not required. The analysis completes in 30 seconds for the new analysis algorithm. It takes 60 seconds, incidentally, for POSET timing to complete. Both algorithms find the same number of states. The analysis reveals that the logic used to compute the stall input at each stage generates glitches. This signal is not critical, however, until the stall latch opens. `ATACS` verifies that this signal is stable before the latch opens and remains stable until the latch closes; thus, it satisfies setup and hold

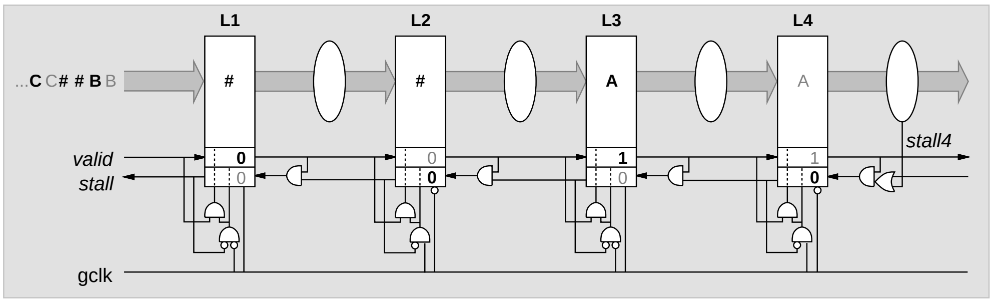


Fig. 6.3. Two-phase interlocked pipeline.

times in normal synchronous operation.

An alternative behavioral model of the pipeline is constructed to see if the glitch can be removed from the design. The logic to compute the stall signal at each stage is moved into the stall latch. The analysis in ATACS shows this design to not produce any glitches on any of its internal wires. The synthesis of this model shows that the latch for the stall signal at each stage is replaced by the generalized C-element shown in Fig. 6.4. This generalized C-element is the latch with the stall logic folded into it. If standard cell design is not an issue, then the internal glitches can be removed using the new gate. The synthesized results also show that the other gates in the design remain unchanged.

6.3.3 Master-slave Interlocked Pipelines

The master-slave implementation of the synchronous interlocked pipeline moves away from two-phase clocking. It uses a one-phase clock and utilizes the fact that most commercial pipeline designs use master-slave latches at each stage. A

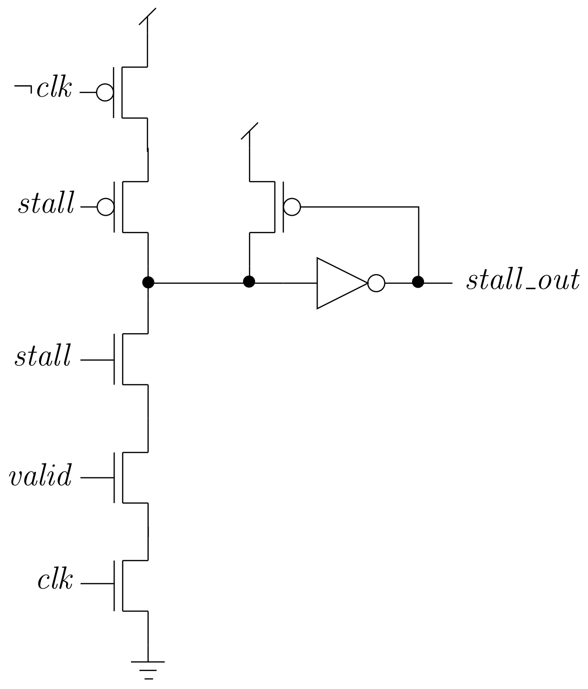


Fig. 6.4. Hazard free circuit for stall signal.

master-slave latch can hold up to two data items. The master half of the latch loads a data item on the rising edge of the clock; and the slave half loads the same data item on the falling edge of the clock. The interlock pipeline gates the clock input to the master and slave portions of the latch. This is shown in Fig. 6.5. This figure resembles the two-phase interlock pipeline in Fig. 6.3, only there is no logic between the master and slave portions of the latch. The master-slave latch can now be used to store two data items to absorb the latency of the stall propagating backward in the pipeline.

The gate level model of two master-slave stages can be analyzed flat on the testing machine, but it takes 700 seconds and there are 116674 reachable states in the gate level specification. The analysis also finds 116685 zones. POSET timing does not complete due to memory limitations. If a module is a single gate in the design, then the average cost of analyzing each gate in a reduced state space is around 14 seconds of running time. There are 36 gates in the specification giving a total cost of 540 seconds to analyze every gate. If a module is the complete master-slave pair with the accompanying logic, then the average cost is 80 seconds for analysis. The complete analysis of the two stage master-slave gate level specification is 160 seconds. This is well under the time to complete the flat analysis.

The analysis shows the design to be correct; although, the *AND* gate between the master and slave latches is once again hazardous. This does not affect the

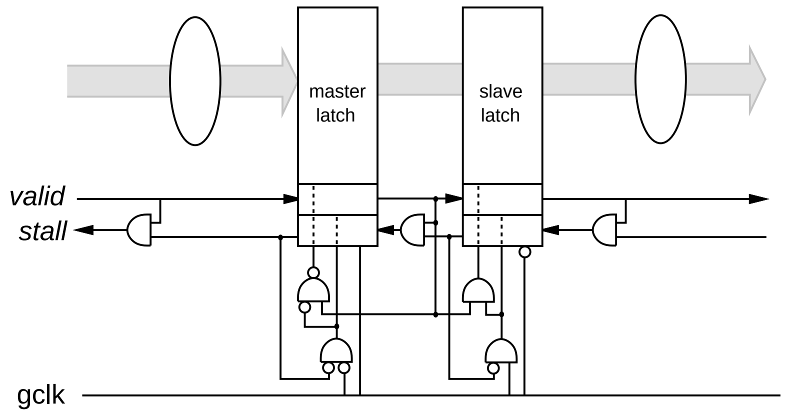


Fig. 6.5. Master-slave interlocked pipeline.

correctness of the master-slave stage because the signal is not an output to the stage. Furthermore, the stall input is not active when the glitch occurs. A new specification is created, however, with the *AND* gate moved into the stall latch. The gate is exactly that shown in Fig. 6.4. This design is shown to not produce any internal glitches because it does not sample the valid and stall signals until the clock edge when both signals are stable.

There is a timing assumption that must be met for the implementation to be correct. The data must be delayed between pipeline stages to prevent race conditions. This is not an issue between the master-slave latches because the master latch is active on the negative phase of the clock; thus, it always opens and closes the master latch after the slave latch. Note that this again is a normal synchronous design constraint.

6.3.4 Fork

A pipeline fork stage copies data to N parallel downstream stages. A fork stage must stall if any of its downstream stages stall. Nonstalled downstream stages must be prevented from receiving duplicate copies of the data when the fork stage is stalled; thus, the valid signals to all downstream stages are lowered until all downstream stall conditions are removed. The data are now received by all downstream stages simultaneously.

The logic for a one to two fork structure in a synchronous interlocked pipeline is shown in Fig. 6.6. The gate level specification of this circuit cannot be analyzed flat by either timing method on the test machine. The fork stage and the two receiving stages are shown to be correct in under 100 seconds of running time using the modular approach. The stall logic is again hazardous when the stall latch is not active. The logic must be moved into the latch to suppress the glitch.

6.3.5 Branch

A pipeline branch stage copies data from an upstream stage to one of N parallel downstream stages. The decision to which of the downstream stages data is to be copied is determined by the datapath logic that generates a set of N one-hot

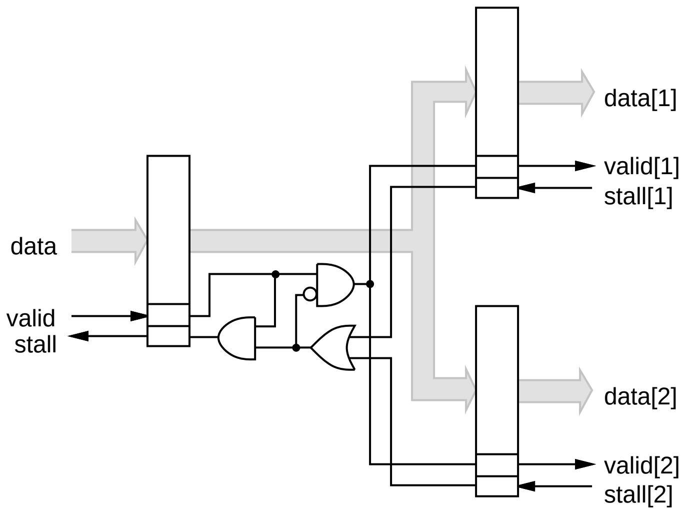


Fig. 6.6. A one to two fork implementation.

encoded enabling signals. These enable signals mask the branch stage valid signal through a set of AND functions such that a valid is propagated only to the selected downstream stage. The branch stage must be stalled only if an already stalled downstream stage is selected as the destination of the data.

The logic for a one to one-of-two branch structure is shown in Fig. 6.7. The gate level specification is too large to analyze flat. The number of states explored in the reduced state space of the left stage in Fig. 6.7 is 114405 zones in 113304 states. The completes in 148Mb of memory with 522 seconds of running time. The total analysis takes around 1600 seconds, but shows the design to be correct.

6.3.6 Join

A pipeline join stage is a merge that concatenates data from N upstream stages to one downstream stage. The join stage waits until data is valid in all upstream stages before concatenating and propagating the data to the downstream stage. A join stage is used to synchronize and align the data streams of multiple pipelines. Any stage that becomes valid must be stalled until all stages have become valid and the data can be propagated to the downstream stage since data in different upstream stages can become valid at different times. If the join stage stalls all upstream stages must stall.

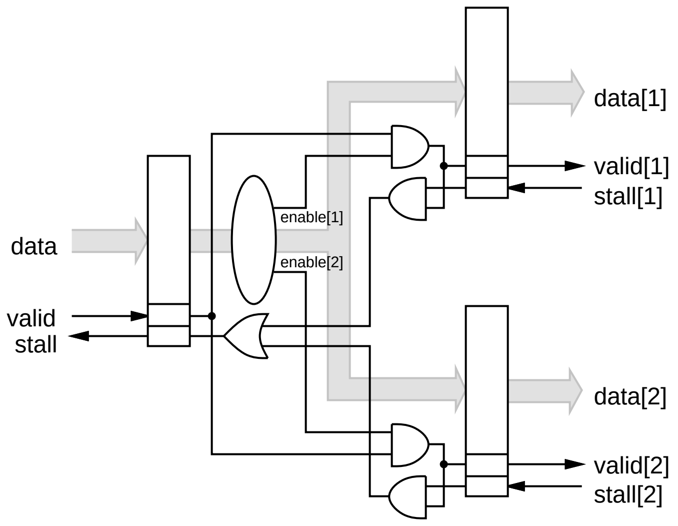


Fig. 6.7. A one to one-of-two branch implementation.

The logic for a two to one join structure is shown in Fig. 6.8. The gate level specification is too large to analyze flat. The number of states explored in the reduced state space of the left top stage in Fig. 6.8 is 11224 zones in 11199 states. This completes in 23Mb with 31 seconds of running time. The total analysis completes in under 120 seconds.

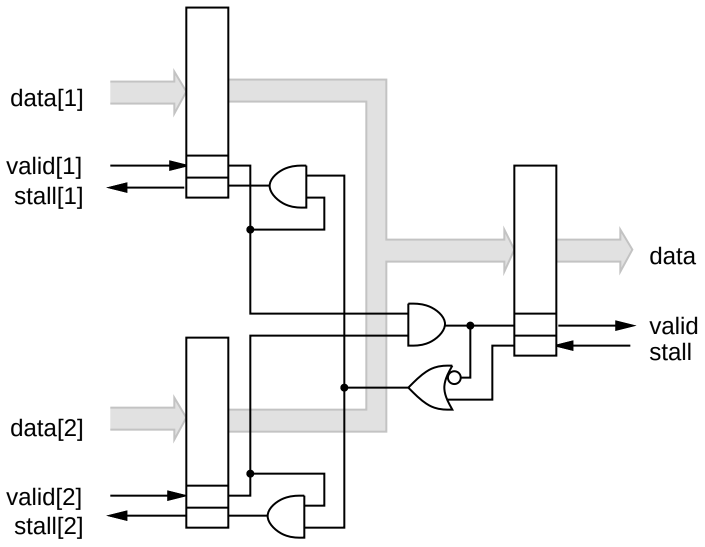


Fig. 6.8. A two to one join implementation.

6.3.7 Priority Select

A pipeline select stage is a selector that propagates data from one of N upstream stages to one downstream stage. A select stage implements a basic if-then-else multiplexer function. A select stage waits until data is valid in at least one of the upstream stages. One stage is then chosen through priority based selection and its data is propagated to the downstream stage. An upstream stage that contains valid data must stall until it is selected.

The logic for a one-of-two to one priority select structure is shown in Fig. 6.9. The gate level specification is too large to analyze flat. The number of states explored in the reduced state space of the top left stage in Fig. 6.9 is 12247 zones in 12241 states. It completes in 23Mb of memory with a running time of 31.17 seconds. The total cost of analysis of all stages is under 120 seconds of running time.

6.4 Summary

The new timing analysis algorithm generally out performs POSET timing for examples that do not use syntactic abstraction. It is competitive with POSET

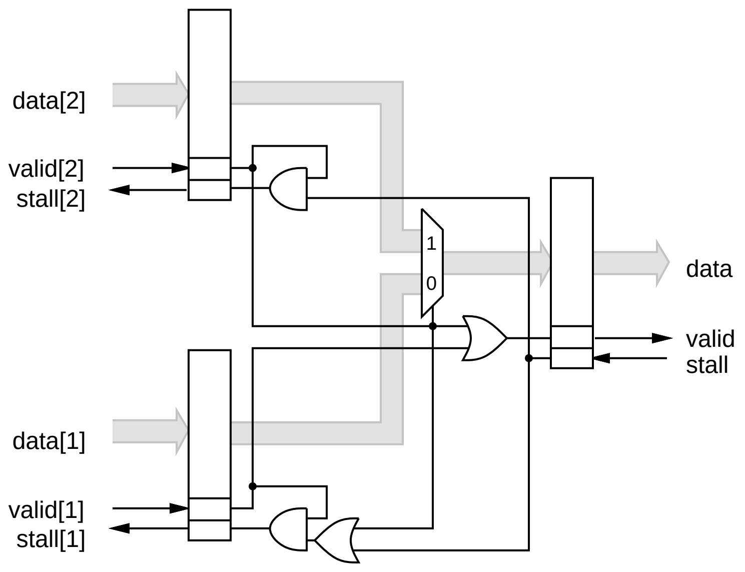


Fig. 6.9. A one-of-two to one priority select implementation.

timing for examples that do use syntactic abstraction. Its performance is largely determined by the effectiveness of the pruning algorithm. The running time is degraded if many transitions cannot be pruned from the zone. This is typically the case for designs where the inputs to every gate are concurrent and causal. The performance on sequential design, such as the synchronous interlocked pipeline, is very positive. The implementation of the new timing analysis algorithm in `ATACS` is more efficient with memory. Memory becomes an issue in the POSET timing algorithm for larger examples; thus, the new analysis algorithm can complete on several examples that are beyond the implementation capacity of POSET timing.

Modular analysis extends the size of specifications that can be analyzed. This is shown in a complete analysis of the synchronous interlock pipeline structures from IBM. The structures are shown to be hazard free at the outputs of each stage. The timing obeys setup and hold times at the latch interfaces. The flat analysis of many of these structures is not possible without a very large compute server. The modular approach makes the application of these algorithms to real designs possible.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The conclusions and future work follow the contributions of the dissertation. It also follows the layout of the dissertation. Each section relates to a contribution. It contains a synopsis of that contribution with a discussion on its impact. That is followed by areas of future work.

7.1 Specification

The level-ruled Petri net is a compact model for timed circuits. It combines the event based Petri net specification with the state machine to produce a model that better represents a circuit. This is demonstrated by its ability to directly model standard and nonstandard designs. This is a key advantage to the level-ruled Petri net. It closely conforms to gate level circuits due to the syntactic abstraction. This makes circuit specification easier to construct by hand, and easier to compile to from higher level languages. The compact representation can improve the performance of analysis algorithms.

The level-ruled Petri net model is different than the timed event/level structure in [43]. The first difference is the semantic and structure basis. The level-ruled Petri net uses the widely accepted Petri net formalism which is a bipartite graph. The timed/event level structure is a directed graph that does not have a notion of a place. This makes the two languages different in their modeling power. The second key difference is seen in conflict. The timed event/level structure uses a relation, where conflict is modeled in the structure of the level-ruled Petri net. This restriction simplifies the semantics and analysis of the level-ruled Petri net.

Not every transition in a specification is relevant or important to correctness. The timing analysis algorithm, however, tracks every signal firing and transition.

Moon adds a notion of *don't care* to the signal transition graph in [35]. This specifies a region of the graph where a signal can arbitrarily toggle. The level-ruled Petri net can benefit from the ability to specify time windows where signal behavior is not important. It is effectively a *don't care* region in time. This can be used to automatically validate setup and hold times in synchronous logic.

The delay model can be greatly improved in the level-ruled Petri net. The minimum/maximum delay model does not accurately reflect a real device. It is not likely that a gate can fire first at its maximum delay and then at it minimum delay on the same die in the same environment. It is also not likely that given two identical gates, right next to each other on the die with the same supply voltage, one gate will switch on its maximum delay with the other switching at its minimum delay. This is the model, however, in the level-ruled Petri net. A correlation factor can greatly improve the delay model. The correlation can be based on location, temperature, mismatch gradient, and other physical realities of the process. The correlation makes the timing model less conservative.

Greenstreet recently published interesting work using the *Charlie diagram* to predict switching delay [78]. The experiment controlled spacing of pulses in an asynchronous timing ring. The Charlie diagram is able to predict if the pulses are evenly spaced or grouped together in a burst as they move around the ring. Applying the Charlie diagram to the level-ruled Petri net as a timing model is interesting work. An important step in the work is validating if a level-ruled Petri net predicts the same timing behavior in the asynchronous ring using the new timing model. This can solidly connect the level-ruled Petri net to physical reality in CMOS design.

7.2 Correctness

The correctness definition is important to understanding the analysis results. This dissertation contributes a formal definition of correctness in the level-ruled Petri net. A net is correct if it is safe, consistent state assigned, output semi-modular, and constraint satisfied. The safe, consistent state assigned, and output semimodular are common correctness properties. Constraint satisfied is becoming more common. The constraint rules in a component are checked in each transition firing to ensure than none of the timers on the rules exceed their latest firing time. Whenever a transition in the component fires, however, the transition's constraint rules are checked to see that they are satisfied in the current state of the system. This enables a designer to specify bounded response and ordering properties in noncausally related transitions.

The current correctness definition only covers safety conditions and bounded response liveness properties. It can check that nothing bad in the model happens, and it can check that something good happens in a bounded amount of time. Future work in correctness needs to address untimed liveness properties. Another issue is the notion of conformance to a specification. Correctness does not consider conformance. It can validate correctness in an environment, but it does not verify that the timed circuit model conforms to a specification. There is no mechanism to compare a gate specification against a behavioral specification. This is an important feature for more general verification.

7.3 Timing Analysis

The timing analysis algorithm for the level-ruled Petri net improves in several examples over previous algorithms. It is difficult, however, to quantify if the improvement is a result of the model or the algorithm. Although the results show favorably for the new algorithm in many cases, it is important to understand that it is being applied to a new model that is different than other models. It is also important to remember that it showed less favorably in certain examples too. The choice of structure and time semantics all impact the cost of timing analysis making some things hard and other things easy. This is readily evident in comparing the new analysis algorithm to algorithms for the time/timed Petri net or the timed automata. There are examples for which the new algorithm out performs existing algorithms, and there are examples showing the opposite. The comparison is not just because the models are so very different.

The new algorithm is a marked improvement over Belluomini in [43]. This is the only published algorithm on an event based model that supports Boolean functions too. POSET timing performs over the new analysis algorithm in a handful of examples, a direct comparison, however, between the two algorithms shows a general reduction from the new analysis algorithm in the size of the timed state space representation on average for a variety of benchmarks. Although the two specification models are different, the improvement is not due to this. It is a result of directly computing causality and pruning out redundant transitions in the zone. These two properties of the new algorithm affect its performance. Although Belluomini presents a similar optimization in [43] for computing causality, it is not general; thus, it cannot match the performance of the new algorithm in all examples.

An important contribution of the new timing analysis algorithm is its ability to support arbitrary Boolean functions. The algorithm in [43] is restricted to conjunctive or disjunctive expressions with a single term in them. This is not the case for the new algorithm; thus, atomic gates that implement complex Boolean functions do not need to be decomposed into two level logic clouds. They can be directly modeled and analyzed as they are implemented. This moves the model and analysis closer to the actual implementation lending credence to the results.

An area of future work is to not generate many small zones in building the finite state space representation. This occurs when there is a small loop in a larger loop that executes until time advances to a certain point. The current algorithm will generate many small zones on each iteration of the loop until the time point is reached. Hendriks presents a method in [79] that avoids generating many small zones by moving directly to the terminating time point. Möller presents a more general approach that accomplishes the same goal [80]. Although these methods are developed for timed automata, they are equally applicable to the level-ruled Petri net. In the given scenario, they have a dramatic affect on running time and representation size.

The timing analysis algorithm needs to use a better time model. A notion of correlation is important to accuracy. It also needs to consider other factors that affect delay such as supply voltage variation, device mismatch, and capacitive coupling. The current timing model is too conservative. It can produce false negatives showing failures where none really exist. A more accurate timing model can alleviate this issue.

The algorithm needs to better support synchronous design. Not all signal transitions matter. The algorithm needs to be able to ignore signal transitions at certain times. For example, signal transitions are not important in a synchronous circuit while a latch is closed. They are important, however, during the setup and hold times for the latch. This type of selective interest is important to modeling timed systems.

Enumerating states is too costly. There are two areas of possible future work in this area. The first is to use symbolic methods to represent the timed state space. The key area of research is in modeling the zone. The second area of research is to change perspective on the verification problem. Rather than starting from an initial state and then searching for a failure, start at a failure and then search for the initial state. For some types of verification issues, this may prove to be very effective.

Another piece of future work is a change of perspective in the algorithm itself. The algorithm currently models the timed state space using zones in the timed state class. These zones contain the time separation on transitions. What if the zones contained the timed separation on rules? This can reduce the cost of computing causality, as well as pruning in the zone. It is also interesting to explore the cost of not adding transitions to the zone as they become enabled. The current algorithm computes casualty multiple times for each transition: once to see if it is fireable, and once to actually fire it; thus, it only explores every combination of causal assignments in a group of marking and level satisfied transitions when it needs to. If many events are removed from the zone due to pruning, however, then there may not be that many combinations to explore in the first place. Adding all enabled transitions into the zone reduces the number of redundant calculations.

7.4 Reduction

Partial order reduction greatly improves the performance of analysis. Large systems can now be analyzed one component at a time. The reduction does not require the model to be altered, and the designer does not have to create a hand abstraction—a process that is often prone to errors. The reduction preserves the exact timed state space of the component. This enables state based synthesis methods to create exact circuits for components in the system. These results show that the idea of applying partial order reduction to the modular synthesis of timed circuits can provide substantial improvements in synthesis time. This includes several examples that could not previously be synthesized using a flat synthesis approach. Future work includes the ability to verify general liveness properties in the reduction.

APPENDIX A

OUTPUT SEMIMODULAR TRANSITIONS

The semimodular property can be verified by looking at groups of reachable states. The groups are created according to the perspective of a circuit seeing only the Boolean state of signals on its interface. All transitions between these state groups must be output semimodular.

Definition A.1 (Boolean Match States). *The pair of states (s, s') are matching states for a set of signals $W' \subseteq W$ if $\nu' \subseteq \nu$ and $(\nu - \nu') \cap W' = \emptyset$ where $s = (\mu, \nu, \mathcal{C})$ and $s' = (\mu', \nu', \mathcal{C}')$.*

Boolean match states look identical from a circuit perspective. They have identical Boolean state valuations for every signal on the circuit's interface. The goal is to build a maximally connected set of Boolean match states. This can be done by a fixed point calculation.

Definition A.2 (One-step reachable). *A pair of state (s, s') is one-step reachable if there exists a transition $t \in T$ and a delay $d \in \mathbb{R}^+ \cup \{\infty\}$ such that either $s \vdash (d, t) \wedge s\,[(d, t)\rangle\, s'$ or $s' \vdash (d, t) \wedge s'\,[(d, t)\rangle\, s$.*

A given pair of states is one-step reachable if some delay-transition pair exists that moves the system from one state to the other. The relation can exist in either direction. The delay-transition pair can move the system from the first state to the second, or it can move the system from the second state to the first. The core function for the fixed point calculation can now be presented using Definition A.1 and Definition A.2.

Definition A.3 (Reach Function). *The reach function $g(S, W')$ returns the set* *of states $S' \supseteq S$ where $s \in S'$ if $s \in S$ or if $s \in [s_o\rangle$ and there exists a state $s' \in S$ such that (s, s') are one-step reachable and Boolean state match on W'.*

The reach function intuitively adds new states to the existing set of states that are reachable by a single delay-transition pair firing and retain a Boolean match with other states already in the set given the signal set W'.

A reduced state graph like that in Fig. A.1(a) can be constructed using the reach function to compute fixed points on the Boolean states defined over a signal set. Each node in the reduced state graph is a set of states that is a Boolean match set for some reduced signal set $W' \subseteq W$.

Definition A.4 (Boolean Match Set). *A set of states S is a Boolean match set for the Boolean state ν' defined over the signals W' if given any member state $(\mu, \nu, \mathcal{C}) \in S$, the following two conditions hold:*

1. *$\nu' \subseteq \nu$ and $(\nu - \nu') \cap W' = \emptyset$; and*
2. *the fixed point $S' = g(S', W')$ is equal to S for any initial seed $S' = \{(\mu, \nu, \mathcal{C})\}$;*

the function $S(\nu', W')$ is the set of all Boolean match sets for ν' and W'; the function $S(W') = \bigcup_{\nu' \in 2^{W'}} \{S(\nu', W')\}$ builds the set of Boolean match sets for all possible Boolean states defined on W'.

The Boolean match set for a given Boolean state and signal set is a maximally connected set of reachable states that share a common state valuation of signals in the signal set W'; all signals not in the signal set are treated as *don't care* signals. Consider again the graph in Fig. A.1(b). Suppose that the signals mapped to t_3 and t_4 are the signal set for the system. The Boolean match set is built from the initial seed $S = \{s_1\}$. The fixed point calculation in the reach function expands the set to $S = \{s_0, s_1, s_2\}$. Each individual state code defined on W' can result in a set of Boolean match sets because a given state code can match in several maximally connected components. The final function $S(W')$ quantifies out each possible Boolean state that can be defined on the signals in W'. Each state set in each member of the $S(W')$ set represents a distinct state perceived in a circuit level implementation. This set is used to validate that a component module has semimodular states. A reduced state graph can now be analyzed to completely validate the output semimodular property for a module in a network of level-ruled Petri nets.

A semimodular state set transition affects transitions moving from one Boolean match set to another. Its counterpart is the semimodular transition. The properties required for a transition to be semimodular in a pair of states must also hold for a transition in a pair of Boolean match sets.

Definition A.5 (Semimodular State Set Transition). *A transition t is semimodular in the state set pair (S_i, S_j) for a given set of visible transitions $T_V \subseteq T$ and a set of transitions on outputs $T_O \subseteq T_V$ if the following implication holds: for all $s_i \in S_i$, $s_j \in S_j$, and $d \in \mathbb{R}^+ \cup \{\infty\}$ such that $s_i \vdash (d, R(t))$ and $s_i\,[(d,t)\rangle\, s_j \implies$*

1. *$t \in T_O \wedge ((\mathsf{mls}(S_i) - \{t\}) \cap T_V \subseteq \mathsf{mls}(S_j) \cap T_V)$; or*

2. *$t \notin T_O \wedge ((\mathsf{mls}(S_i) - \{t\}) \cap T_O \subseteq \mathsf{mls}(S_j) \cap T_O)$;*

where for a given $s = (\mu, \nu, \mathcal{C})$, $\mathsf{mls}(s) = \{t \in T \mid (\mu, \nu) \vdash R(t)\}$, and for a given state set S, $\mathsf{mls}(S) = \bigcup_{s \in S} \mathsf{mls}(s)$.

This definition considers all marking and level satisfied transitions in the two state sets to determine the semimodular property. The first part of the implication exists to filter out transitions and state set pairs that are not connected. These are semimodular by default since the transition does not fire in a state in one state set to move the system to a state in the other state set. If, however, such a state pair exists for the two state sets, then one of the two stated conditions must hold. If the transition is an output from T_O, the first condition does not allow the firing of the output transition to disable any marking and level satisfied visible transitions in the states of the S_i set. If the transition is not an output, then the second condition does not allow the firing of the transition to disable any marking and level satisfied transitions in the states of the S_i set. These conditions exactly match those for an output semimodular transition in Definition 3.13, only these consider all marking and level satisfied transition from the states in the entire state sets S_i and S_j.

The semimodular state set transition validates semimodular states for a component. Consider again the fragment of the state graph in Fig. A.1(b). The Boolean match sets for the component are shown in Fig. A.1(a). The transition t_3 is not an output semimodular state set transition by the second condition of Definition A.5 for $S = \{s_0, s_1, s_2\}$ and $S' = \{s_3\}$ given $T_V = \{t_3, t_4\}$ and $T_O = \{t_4\}$. From the level-ruled Petri net in Fig. A.1(c), firing the input transition t_3 disables the output t_4; thus, the marking and level satisfied set from S minus t_4 is $\{t_3\}$ and the same set is empty for S'. At the circuit level perspective, this module does not have semimodular states.

Definition A.6 (Output Semimodular). *A component M_i in a network of level-ruled Petri nets $M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$ is output semimodular if two conditions hold for all transitions $t \in T$:*

1. *for all state pairs $(s, s') \in [s_o\rangle$, there exists a delay $d \in \mathbb{R}^+ \cup \{\infty\}$ such that $s_i \vdash (d, R(t))$ and $s_i\,[(d,t)\rangle\, s_j \implies t$ is a semimodular transition on (s, s') given T_V and T_O; and*

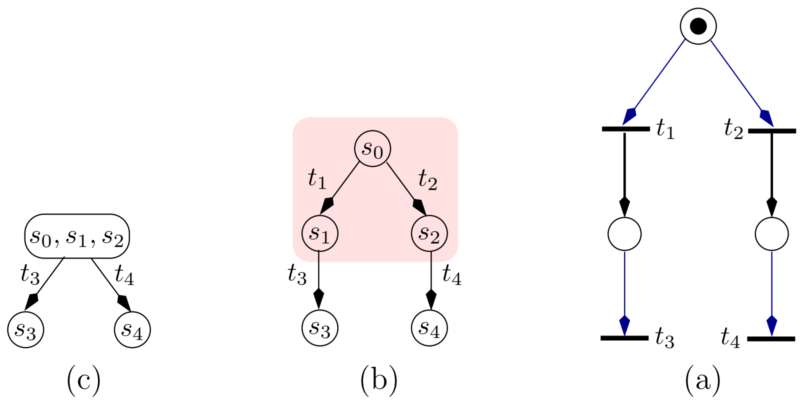


Fig. A.1. A net fragment with its full and reduced state space. (a) A reduced state graph for the visible transitions t_3 and t_4 showing the state semimodular violation. (b) The state graph for the net fragment. (c) A level-ruled Petri net fragment that enables two invisible transitions t_1 and t_2 on a choice place eventually followed by the visible transitions t_3 and t_4.

2. *for all state set pairs* $(S, S') \in S(W_i)$*,* t *is a semimodular state set transition given* T_V *and* T_O*.*

where $T_V = T(W_i)$ *and* $T_O = T(O_i)$*.*

If there does not exist a delay such that the delay-transition pair is enabled in the first state, and firing it leads to the second state, then the transition is semimodular because it cannot fire from the first state. If there does exist a delay such that the delay-transition pair is enabled and firing it from the first state results in the second state, then one of two conditions must hold for the transition to be semimodular. The first condition in Definition A.6 ensures that the transitions in a component model are semimodular. Many output semimodular violations are discovered by the first condition. More importantly, the first condition can be dynamically validated as a firing sequence is evolved. The second condition in Definition A.6 ensures that the module has output semimodular states at the circuit perspective. It covers all of the violations in the first condition too, but it cannot be dynamically applied to an evolving firing sequence. It exists to discover output semimodular state violations that can not be uncovered in an evolving firing sequence; thus, both the reachable state set and allowed firing sequences are required to validate that a module is output semimodular in a defined environment.

APPENDIX B

BOOLEAN FUNCTION MANIPULATION

The level necessary for a transition is derived from its rule set. It is the prime implicants of the function formed by the conjunction of the functions on each rule in the rule set of the transition in max term form mapped onto transitions. The maxterm prime implicants can be computed with known algorithms. This section presents operations on the Lsat in the level-ruled Petri net to compute the prime implicants recursively using positive and negative cofactors. The recursive algorithm itself is not presented.

A Boolean state of the system is a member of the power set of the signals it is defined over. This is any $\nu \in 2^W$ for the signal set W in the level-ruled Petri net. For any signal $w \in W$ and Boolean state $\nu \in 2^W$, $w \in \nu$ means that the signal is in its high Boolean state, and $w \notin \nu$ means that the signal is in its low Boolean state; thus, the Boolean state $\{a, b\}$ defined over the signal set $\{a, b, c\}$ is understood to mean that the signals a and b are high, and the signal c is low. Recall from Definition 2.10 that the Lsat member of the level-ruled extension implements the syntactic abstraction. It is the function $\mathsf{Lsat} : R \rightarrow (2^W \rightarrow \{\mathsf{true}, \mathsf{false}\})$. The function takes a rule and a Boolean state defined over the signals in the level-ruled Petri net, and it returns either true or false. The true value indicates that the level information is satisfied by the passed in state vector; the false value indicates the opposite.

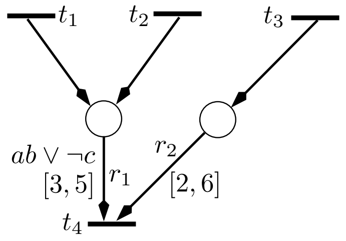
A Boolean function for a rule has a canonical form that can be a set of minterms or a set of maxterms. A minterm is a product term in a Boolean function, and it is a Boolean state where the function evaluates to true.

Definition B.1 (Sum of Products Form). *The sum of products canonical form* *of a Boolean function defined over the set of signals W for a given rule $r \in R$ is $f(r) = \left\{\nu \in 2^W \mid \mathsf{Lsat}(r)(\nu) = \mathsf{true}\right\}$; a function is always* true *if $f(r) = 2^W$; a function is always $false$ if $f(r) = \emptyset$; the sum of products form for a transition $t \in T$ is $f(t) = \bigcap_{r \in R(t)} f(r)$.*

The sum of products form of the Boolean function is the set of minterms for the function. It is only necessary to satisfy a single minterm for the function to be true. Consider again the rule r_1 in Fig. B.1 with the Boolean function $ab \vee \neg c$ defined over the set $W = \{a, b, c\}$. The sum of product form for r_1 is the set $f(r_1) = \{\emptyset, \{b\}, \{a\}, \{a, b\}, \{a, b, c\}\}$. Using a more standard notation, $f(r_1) = (\neg a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) \vee (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$. This notation is used for the rest of the presentation. The sum of products form of the Boolean function for the transition t_4 is $f(t_4) = f(r_1)$ because $f(r_2) = \mathsf{true}$.

The sum of product form with the negative and positive cofactor is used to compute the dependence of a Boolean function on a given signal. Suppose that there exists a transition t such that $f(t) = (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$. The signal c does not affect the truth value of this function because it is level satisfied whenever a and b are high. This is detected using cofactor operations.

Definition B.2 (Positive Cofactor). *The positive cofactor on a signal $w \in W$ in a Boolean function f in the sum of products form defined over the set of signals W is $f_w = \left\{\nu \in 2^W \mid \exists \nu' \in f : w \in \nu' \wedge (\nu = \nu' - \{w\}) \vee (\nu = \nu')\right\}$; $f_w(t)$ returns the positive cofactor of the signal w in the sum of products form of the Boolean function for the transition $t \in T$.*



$$L(t_5) = a+$$
$$L(t_6) = b+$$
$$L(t_7) = c-$$

Fig. B.1. A fragment with a merge place and a rule with a Boolean function.

The positive cofactor of a function is all the minterms in the function with the signal w removed if it originally appeared in the minterm or the minterm itself if signal w is appears in it. This makes signal w a *don't care* term in the sum of products representation. The positive cofactor on c for the transition in this example is $f_c(t) = (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$.

Definition B.3 (Negative Cofactor). *The negative cofactor on a signal $w \in W$ in a Boolean function f in the sum of products form defined over the set of signals W is $f_{\neg w} = \{\nu \in 2^W \mid \exists \nu' \in f : w \notin \nu' \wedge (\nu = \nu' \cup \{w\}) \vee (\nu = \nu')\}$; $f_{\neg w}(t)$ returns the negative cofactor on the signal w in the sum of products form of the Boolean function for the transition $t \in T$.*

The negative cofactor is similar to the positive only it adds new minterms containing the signal w rather than adding minterms that do not contains the signal w like the positive cofactor. The negative cofactor on c for the example transition is $f_{\neg c}(t) = (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$. The signal c is independent of the Boolean function for t in this example because $f_c(t) = f_{\neg c}(t)$. This is not the case for the signal c and the transition t_4 in Fig. B.1. The positive cofactor on c is $f_c(t_4) = (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$, and the negative cofactor on c is $f(t_4) = (\neg a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge b \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (a \wedge \neg b \wedge c) \vee (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$ This function depends on the signal c, and more specifically, it depends on $\neg c$ because it is negative unate; this means that $f_{\neg c}(t_4) \supseteq f_c(t_4)$. This is necessary information in computing a causal group set.

Not all signals contribute to the truth value of the function as shown above where $f_c(t_4) = f_{\neg c}(t_4)$. The level necessary set needs to be in terms of transitions that contribute to the truth value of the function. The positive and negative cofactor are used to test if a signal is independent in a function—meaning that it does not contribute to the function's truth value. Recall the example of $f(t) = (a \wedge b \wedge \neg c) \vee (a \wedge b \wedge c)$ for some transition $t \in T$. The signal c is independent in this function because $f_c(t) = f_{\neg c}(t)$. The Boolean function for t_4 in Fig. B.1, however, depends on the negative phase of c because $f_{\neg c}(t_4) \supseteq f_c(t_4)$.

Definition B.4 (Level Transition Set). *The set of level transitions for a transition $t \in T$ is the set of transitions $t' \in T$ such that there exists a signal $w \in W$ where $f_w(t) \neq f_{\neg w}(t)$ (not independent) and one of the following holds:*

1. $f_w(t) \supseteq f_{\neg w}(t) \wedge L(t') = w+$ *(positive unate);*

2. $f_w(t) \subseteq f_{\neg w}(t) \wedge L(t') = w-$ *(negative unate); or*

3. $f_w(t) \not\supseteq f_{\neg w}(t) \wedge f_w(t) \not\subseteq f_{\neg w}(t) \wedge (L(t') = w+ \vee L(t') = w-)$ *(mixed unate);*

the function $\mathsf{lts}(t)$ *returns the level transition set for the transition t;* $\mathsf{lts}(r)$ *returns the level transition for a rule $r \in R$.*

The level transition set is the set of transitions on signals that affect the truth value of the Boolean function on the given transition. If the signal is not independent in the function, then there are three cases to consider: the first case indicates the function is positive unate on the signal; the second case indicates that the function is negative unate on the signal; and the third case indicates that the function is mixed unate on the signal. Consider a transition $t \in T$ with an exclusive-or function defined over the signals set $W = \{a, b\}$. The sum of products form is $f(t) = (a \wedge \neg b) \vee (\neg a \wedge b)$. The positive cofactor on a is $f_a(t) = (a \wedge \neg b) \vee (\neg a \wedge \neg b)$, and the negative cofactor on a is $f_{\neg a}(t) = (\neg a \wedge b) \vee (a \wedge b)$. This satisfies the third case where the function depends on the positive and negative phase of the signal a because the positive and negative cofactor on a are not related. Consider the transition t_4 in Fig. B.1 with the specified transitions t_5, t_6, and t_7. The level transition set for t_4 is $\{t_5, t_6, t_7\}$.

The level necessary set for a transition t is essentially the collection of Boolean function created from each rule $r \in R(t)$ in its product of sums form. A product of sums form is a set of maxterms. A maxterm is a sum term in a Boolean function. It is a set of disjunctive signals. A single signal in the maxterm must be in its specified Boolean state to satisfy the sum term. The product of sums Boolean function representation is a set of maxterms for the function.

Definition B.5 (Product of Sums Form). *The product of sums canonical form of a Boolean function defined over the set of signals W for a given rule $r \in R$ is $\breve{f}(r) = \left\{\nu \in 2^W \mid \exists \nu' \in 2^W : \nu = W - \nu' \wedge \mathsf{Lsat}(r)(\nu') = \mathsf{false}\right\}$; a function is always* false *if $\breve{f}(r) = 2^W$; a function is always* true *if $\breve{f}(r) = \emptyset$; the product of sums form for a transition $t \in T$ is $\breve{f}(t) = \bigcup_{r \in R(t)} \breve{f}(r)$.*

A maxterm is the compliment of the signals in a state where the function evaluates to false. This is computed by $W - \nu'$ where ν' is a state where the function is false. The set of all these maxterms is the product of sums form for the Boolean function. Consider again the rule r_1 in Fig. B.1 with the Boolean function $ab \vee \neg c$ defined over the set $W = \{a, b, c\}$. The product of sums form for r_1 is the set $\breve{f}(r_1) = (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee b \vee \neg c)$. The product of sums form for the rule r_2 is $\breve{f}(r_2) = \mathsf{true}$, because the function is true in every state by definition. The product of sums form of the Boolean function for the transition t_4 is $\breve{f}(t_4) = \breve{f}(r_1)$ because $\breve{f}(r_2) = \mathsf{true}$.

The level transition set with the product of sums form of the Boolean functions is used to build the final level necessary set for a given transition. The set contains only transitions that affect the truth value of the function.

Definition B.6 (Level Necessary Set). *The level necessary set $\mathsf{lrs}(t)$ for a given transition $t \in T$ is $\mathsf{lrs}(t) = \bigcup_{\nu \in \breve{f}(t)} \left\{\bigcup_{w \in W} x(t, \nu, w)\right\}$; where the function $x(t, \nu, w)$ returns the set of transitions from the level transition set that must fire for the signal w to match its phase in the Boolean state ν; this is given as $x(t, \nu, w) = \{t' \in T \mid t' \in \mathsf{lts}(t) \wedge ((w \in \nu \wedge L(t') = w+) \vee (w \notin \nu \wedge L(t') = w-))\}$.*

Consider again the transition t_4 in Fig. B.1 with the specified transitions t_5, t_6, and t_7 for $a+$, $b+$, and $c-$, respectively. Recall that the product of sums form for the Boolean function on t_4 is $\breve{f}(t_4) = (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee b \vee \neg c)$. The level transition set for t_4 is $\{t_5, t_6, t_7\}$ so $\neg a$ and $\neg b$ are independent in the function. The level necessary set for t_4 is $\{\{t_5, t_6, t_7\}, \{t_5, t_7\}, \{t_6, t_7\}\}$. This is a result of the Boolean function on t_4. The $\{t_5, t_7\}$ set comes from the $(a \vee \neg b \vee \neg c)$ maxterm; the $\{t_6, t_7\}$ derives from the $(\neg a \vee b \vee \neg c)$ maxterm; finally, the $\{t_5, t_6, t_7\}$ set comes from the $(a \vee b \vee \neg c)$ maxterm. Note that if there existed another transition t_8 in this system such that $L(t_8) = a+$, then the level necessary set includes both t_5 and t_8 for any $a+$ transition; thus the new level necessary set for t_4 considering this new transition is $\{\{t_5, t_6, t_7, t_8\}, \{t_5, t_7, t_8\}, \{t_6, t_7\}\}$.

The level necessary set given in Definition B.6 is not optimal. The optimal level necessary set is the set of prime implicants for the Boolean function defined in a transition. The level necessary set in Definition B.6 may be larger than it needs to be, but it is still correct nonetheless; it produces the same set of causal groups as the possible smaller set created from the prime implicants. It is less optimal because it can result in a larger finite representation of the reachable state space. The level necessary set is key to computing new timed state classes from firing transitions using different causal groups. Although the causal groups are the same, a larger level necessary set implies smaller zones in the timed state class; thus, more zones are required to cover the state space. The Boolean functions on each transition derived from its rule set that are used to create the level necessary sets can come from a logic optimizer, be provided by the user for each transition, or be constructed from the maxterms in the Boolean functions of the rules. The smallest set is the most optimal.

REFERENCES

[1] K. S. Stevens, S. Rotem, R. Ginosar, P. A. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken, “An asynchronous instruction length decoder,” *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 217–228, Feb. 2001.

[2] I. Sutherland and S. Fairbanks, “GasP: A minimal FIFO control,” in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 46–53, IEEE Computer Society Press, Mar. 2001.

[3] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins, “Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 ghz,” in *Proc. International Solid State Circuits Conf*, 2000.

[4] H. P. Hofstee, S. H. Dhong, D. Meltzer, K. J. Nowka, J. A. Silberman, J. L. Burns, S. D. Posluszny, and O. Takahashi, “Designing for a gigahertz,” *IEEE MICRO*, May-June 1998.

[5] C. A. Petri, “Communication with automata,” Tech. Rep. RADC-TR-65-377, Vol. 1, Suppl 1, Applied Data Research, Princeton, NJ, 1966.

[6] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[7] T. Murata, “Petri nets: Properties, Analysis, Applications,” *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.

[8] C. J. Myers, *Asynchronous Circuit Design.* John Wiley & Sons, NY, July 2001.

[9] H. Zheng, “Specification and compilation of timed systems,” Master’s thesis, University of Utah, 1998.

[10] W. Belluomini, C. J. Myers, and H. P. Hofstee, “Timed circuit verification using TEL structures,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 20, pp. 129–146, Jan. 2001.

[11] D. E. Muller and W. S. Bartky, “A theory of asynchronous circuits,” in *Proceedings of an International Symposium on the Theory of Switching*, pp. 204–243, Harvard University Press, Apr. 1959.

[12] H. Jacobson, P. Kudva, P. Bose, P. Cook, S. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked piplines," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 3–12, IEEE Computer Society Press, Apr. 2002.

[13] W. Belluomini, C. J. Myers, and H. P. Hofstee, "Verification of delayed-reset domino circuits using ATACS," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 3–12, IEEE Computer Society Press, Apr. 1999.

[14] M. R. Greenstreet, "Private communication," 2002.

[15] I. Sutherland and J. Ebergen, "Computers without clocks," in *Scientific American*, Aug. 2002.

[16] M. R. Greenstreet, "Implementing a STARI chip," in *Proc. International Conf. Computer Design (ICCD)*, pp. 38–43, IEEE Computer Society Press, 1995.

[17] M. R. Greenstreet, "Stari: Skew tolerant communication." unpublished manuscript, 1997.

[18] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[19] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic Model Checking for Real-Time Systems," in *7th. Symposium of Logics in Computer Science*, (Santa-Cruz, California), pp. 394–406, IEEE Computer Scienty Press, 1992.

[20] M. Bozga, O. Maler, A. Pnueli, and S.Yovine, "Some progress in the symbolic verification of timed automata," in *Proc. 9th International Conference on Computer Aided Verification* (O. Grumberg, ed.), vol. 1254, pp. 179–190, Springer Verlag, 1997.

[21] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The tool KRONOS," in *Hybrid Systems III: Verification and Control*, vol. 1066, (Rutgers University, New Brunswick, NJ, USA), pp. 208–219, Springer, 22–25 October 1995.

[22] O. Maler and A. Pnueli, "Timing analysis of asynchronous circuits using timed automata," in *Correct Hardware Design and Verification Methods* (P.E. Camurati and H. Eveking, eds.), vol. 987, (Spectre-Verimag (France), Weizmann Inst. (Israel)), pp. 189–205, Springer-Verlag, 1995.

[23] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems," in *Hybrid Systems*, pp. 232–243, 1995.

[24] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial order reductions for timed systems," in *International Conference on Concurrency Theory*, pp. 485–500, 1998.

[25] M. Minea, *Partial order reduction for verification of timed systems.* PhD thesis, Carnegie Mellon University, School of Computer Science, 1999.

[26] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada* (A. J. Hu and M. Y. Vardi, eds.), vol. 1427, pp. 546–550, Springer-Verlag, 1998.

[27] M. Bozga, O. Maler, and S. Tripakis, "Efficient verification of timed automata using dense and discrete time semantics," in *Conference on Correct Hardware Design and Verification Methods*, pp. 125–141, 1999.

[28] M. Bozga, H. Jianmin, O. Maler, and S. Yovine, "Verification of asynchronous circuits using timed automata," in *Electronic Notes in Theoretical Computer Science* (O. M. Eugene Asarin and S. Yovine, eds.), vol. 65, Elsevier Science Publishers, 2002.

[29] T. Chu, "On the models for designing VLSI asynchronous digital circuits," *Integration, the VLSI journal*, vol. 4, pp. 99–113, June 1986.

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

[31] L. Y. Rosenblum and A. V. Yakovlev, "Signal graphs: from self-timed to timed ones," in *Proceedings of International Workshop on Timed Petri Nets*, (Torino, Italy), pp. 199–207, IEEE Computer Society Press, July 1985.

[32] C. L. Seitz, "Asynchronous machines exhibiting concurrency," 1970. Record of the Project MAC Concurrent Parallel Computation.

[33] C. E. Molnar, T. Fang, and F. U. Rosenberger, "Synthesis of delay-insensitive modules," in *1985 Chapel Hill Conference on Very Large Scale Integration* (H. Fuchs, ed.), pp. 67–86, Computer Science Press, 1985.

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

[35] C. W. Moon, P. R. Stephan, and R. K. Brayton, "Synthesis of hazard-free asynchronous circuits from graphical specifications," in *Proc. International Conf. Computer Design (ICCD)*, pp. 322–325, IEEE Computer Society Press, Nov. 1991.

[36] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed Petri nets," Tech. Rep. Project MAC Tech. Rep. 120, Massachusetts Inst. of Tech., Feb. 1974.

[37] P. Merlin and D. J. Faber, "Recoverability of communication protocols," *IEEE Transactions on Communications*, vol. 24, no. 9, 1976.

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

[39] 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.

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

[41] S. M. Burns and A. J. Martin, "Performance analysis and optimization of asynchronous circuits," in *Advanced Research in VLSI*, pp. 71–86, MIT Press, 1991.

[42] C. J. Myers, *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Stanford University, 1995.

[43] W. Belluomini, *Algorithms for Synthesis and Verification of Timed Circuits and Systems.* PhD thesis, University of Utah, 1999.

[44] S. Tasiran and R. K. Brayton, "Stari: A case study in compositional and heirarchical timing verification," in *Proc. International Conference on Computer Aided Verification*, 1997.

[45] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pp. 195–220, 1982.

[46] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Transactions on Computers*, vol. 35, pp. 1035–1044, Dec. 1986.

[47] D. L. Dill and E. M. Clarke, "Automatic verification of asynchronous control circuits using temporal logic," in *1985 Proceedings at Chapel Hill Conference on VLSI* (H. Fuchs, ed.), pp. 127–143, Computer Science Press Inc., 1985.

[48] E. M. Clarke and A. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications," in *ACM Transactions on Programming Langauges and Systems*, pp. 244–263, Feb. 1986.

[49] J. R. Burch, "Combining ctl, trace theory and timing models," in *Proceedings of the First Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[50] R. Alur, C. Courcoubetis, and D. L. Dill, "Model-checking in dense real-time," in *Information and Computation 103*, May 1992.

[51] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarke, "Efficient verification of parallel real-time systems," in *Computer Aided Verification* (C. Courcoubetis, ed.), pp. 321–332, Springer-Verlag, 1993.

[52] R. Alur and R. P. Kurshan, "Timing analysis in cospan," in *Hybrid Systems III*, Springer-Verlag, 1996.

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

[54] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. Nowick, "A correctness criterion for asynchronous circuit validation and optimization," *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 1309–1318, Nov. 1994.

[55] J. R. Burch, "Modeling timing assumptions with trace theory," in *Proc. International Conf. Computer Design (ICCD)*, 1989.

[56] T. Yoneda and H. Ryu, "Timed trace theoretic verification using partial order reduction," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 108–121, IEEE Computer Society Press, Apr. 1999.

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

[58] B. Zhou, T. Yoneda, and C. Myers, "Framework of timed trace theoretic verification revisited," *The Institute of Electronics, Information, and Communication Engineers Transactions*, 2002.

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

[60] R. Camposano, S. Devadas, K. Keutzer, S. Malik, and A. Wang, "Implicit enumeration techniques applied to asynchronous circuit verification," in *Proc. Hawaii International Conf. System Sciences*, vol. I, IEEE Computer Society Press, Jan. 1993.

[61] H. Hulgaard, *Timing Analysis and Verification of Timed Asynchronous Circuits.* PhD thesis, Department of Computer Science, University of Washington, 1995.

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

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

[64] S. Tasiran, R. Alur, R. Kurshan, and R. Brayton, "Verifying abstractions of timed systems," in *LNCS*, vol. 1119, pp. 546–562, Springer-Verlag, 1996.

[65] C. J. Myers, T. G. Rokicki, and T. H. Y. Meng, "POSET timing and its application to the synthesis and verification of gate-level timed circuits," *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 769–786, June 1999.

[66] W. Belluomini and C. J. Myers, "Timed state space exploration using posets," *IEEE Trans. Computer-Aided Design*, vol. 19, May 2000.

[67] A. Valmari, "A stubborn attack on state explosion," in *International Conference on Computer-Aided Verification*, pp. 176–185, June 1990.

[68] P. Godefroid, "Using partial orders to improve automatic verification methods," in *International Conference on Computer-Aided Verification*, pp. 176–185, June 1990.

[69] T. Yoneda, E. G. Mercer, and C. J. Myers, "Modular synthesis of timed circuits using partial order reduction," in *Proceedings of International Workshop on Synthesis and System Integration of Mixed Technologies*, Oct. 2001.

[70] A. Semenov, A. Yakovlev, E. Pastor, M. P. na, J. Cortadella, and L. Lavagno, "Partial order based approach to synthesis of speed-independent circuits," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 254–265, IEEE Computer Society Press, Apr. 1997.

[71] E. G. Mercer, C. J. Myers, and T. Yoneda, "Improved POSET timing analysis in Timed Petri Nets," in *Proceedings of International Workshop on Synthesis and System Integration of Mixed Technologies*, October 2001.

[72] H. Zheng, E. G. Mercer, and C. J. Myers, "Automatic abstraction for verification of timed circuits and systems," *Lecture Notes in Computer Science*, vol. 2102, pp. 182–193, 2001.

[73] P. A. Beerel, T. H.-Y. Meng, and J. Burch, "Efficient verification of determinate speed-independent circuits," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, pp. 261–267, IEEE Computer Society Press, Nov. 1993.

[74] P. A. Beerel, J. R. Burch, and T. H.-Y. Meng, "Checking combinational equivalence of speed-independent circuits," *Formal Methods in System Design*, Mar. 1998.

[75] O. Roig, *Formal Verification and Testing of Asynchronous Circuits.* PhD thesis, Univsitat Politècnia de Catalunya, May 1997.

[76] V. Vakilotojar and P. Beerel, "Hiding memory elements in induced hierarchical verification of speed-independent circuits," in *Proc. International Workshop on Logic Synthesis*, June 1998.

[77] C. J. Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng, "Timed circuits: A new paradigm for high-speed design," in *Proc. of Asia and South Pacific Design Automation Conference*, pp. 335–340, Feb. 2001.

[78] A. Winstanley, A. Garivier, and M. Greenstreet, "An event spacing experiment," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 47–56, IEEE Computer Society Press, Apr. 2002.

[79] M. Hendriks and K. G. Larsen, “Exact acceleration of real-time model checking,” *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 7, 2002.

[80] M. O. Möller, “Parking can get you there faster,” *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 7, 2002.