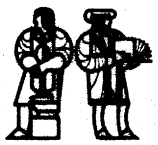
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-393

SYNTHESIS OF SELF-TIMED VLSI CIRCUITS FROM GRAPH-THEORETIC SPECIFICATIONS

Tam-Anh Chu

June 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications

by

Tam-Anh Chu

B.S.E.E. Catholic University, Washington, D.C. (1979)

S.M. Massachusetts Institute of Technology (1981)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy
at the
Massachusetts Institute of Technology

June, 1987

© Tam-Anh Chu

The author hereby grants to M.I.T. permission to reproduce and
to distribute copies of this thesis document in whole or in part

Signature of Author _____

Department of Electrical Engineering and Computer Science
May 11, 1987

Certified by _____

Jack B. Dennis
Thesis Supervisor

Accepted by _____

Arthur C. Smith
Chairman, Department Committee on Graduate Students

Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications

by
Tam-Anh Chu

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 1987 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

This thesis presents an approach for direct and efficient synthesis of self-timed (asynchronous) control circuits from formal specifications called *Signal Transition Graphs* (STGs). Control circuits synthesized from this graph model are *speed-independent* and capable of perfoming concurrent operation. The property of speed-independence means that the circuit operates correctly regardless of variations in delays of logic gates, thus implying that the circuit is hazard-free under any combination of gate delays. The capability of STGs for explicitly specifying concurrent operations internal to a control circuit is unique to this model, unlike other approaches based on Finite State Machines.

STGs are a form of interpreted Petri nets, in which transitions in a net are interpreted as transitions of signals in a control circuit. While other synthesis approaches based on Petri nets have not been very successful, we have developed a number of analytical results which establish the equivalence between the static structure of nets (their syntax) and their underlying firing sequence semantics—an analytical approach called *structure theory* of Petri nets. This equivalence permits the characterization of the low-level properties of control circuits in terms of STG syntax: the properties of deadlock-free and hazard-free of circuits are characterized as syntactic propeties of *liveness* and *persistency* of STGs. A preliminary STG specification of a control circuit can be modified into one which is live and persistent, from which a deadlock-free and hazard-free logic implementation can be derived mechanically.

STGs allow efficient synthesis of control circuits by using a method of decomposition based on a graph-theoretic technique called *contraction*. Instead of implementing a logic circuit from a STG directly, it can first be decomposed into a number of contracted nets, one for each signal generated by the control circuit. A logic element can then be determined from each contracted net, and the composition of logic elements produces the final circuit implementation.

Thesis supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering

Keywords: Asynchronous, self-timed speed-independent circuits; VLSI; Petri nets; structure theory; concurrency; finite automata.

Acknowledgments

I would like to thank Jack Dennis, my thesis supervisor, for providing invaluable guidance during the course of my thesis research. He had helped me formalize my ideas and guided me through the maze of my own confusion. I have learned a great deal about research, among many other things, from him.

Lance Glasser and Bill Dally, my thesis readers, deserve a lot of credits in shaping my research. Lance taught me VLSI circuit design, and the importance of always reasoning from first principles. Bill provided me with many useful comments and fresh perspectives about the problem.

Special thanks to my friend Clement Leung who has over the years given me advice and guidance; his friendship and help have made my life much easier. The ideas in this thesis originated from our early collaboration, and he deserves a share of them.

Many present and past members of the Computation Structures Group have helped me and made my stay at MIT more fun; I would like to mention a few: Arvind, Bill Ackerman, Andy Boughton, Suresh Jagannathan, Willie Lim, Guang-Rong Gao, Suhas Patil, Natalie Tarbet, Kevin Theobald, Tom Wanuga, Earl Waldin.

My Vietnamese friends at MIT and Harvard have provided me with a second home away from home, and I appreciate their support. Among them: Minh Hoàng for being a good old buddy; Bích-Ngọc Trần for her friendship, and also for her proofreading part of the early draft of this thesis.

This thesis could not have been completed without the love and support of my parents, my brother Nhị-Anh and sisters Nhất-Anh and Tứ-Anh. To them, I would like to dedicate this thesis.

The major part of this research was carried out under financial assistance from the Hughes Doctoral Fellowship Program. This document was prepared using LaTeX and equipments at the MIT Laboratory for Computer Science.

Contents

1 Introduction 1

1.1 Objectives 1

1.2 Background 2

1.3 Problems with some previous models 5

1.4 Main contributions of this work 7

1.5 Experimental results 8

1.6 Organization of thesis 11

2 An Informal Introduction 16

2.1 Petri Nets and Signal Transition Graphs 18

2.1.1 Petri nets 18

2.1.2 Signal Transition Graphs 23

2.1.3 Relation between Petri nets and Signal Transition Graphs 25

2.2 Semantics of nets/Behavioral equivalence 27

2.2.1 Behavioral Equivalence 28

2.2.2 Important Analytical Results for Nets 29

2.3 Properties of State graphs 34

2.3.1 State Graphs and Network Functions 34

2.3.2 Liveness and Consistent State Assignment 35

2.3.3 Persistency 36
2.3.4 A synthesis procedure 38
2.4 Decomposition by Net Contraction 39
2.5 A problem with state assignment 44
2.6 Summary 45

3 Semantics and Temporal Relations of Nets **46**
3.1 Petri Nets 47
3.1.1 A Brief Introduction 47
3.1.2 Previous Results for Free-Choice nets 50
3.2 Firing Sequence Semantics of Free-Choice nets 53
3.2.1 Semantics 53
3.2.2 An algorithm for constructing reachability graphs 55
3.3 Temporal Relations: Ordering, Concurrency and Conflict 58
3.3.1 Syntactic Characterization 58
3.3.2 Partition of the Temporal Relation and Correspondence to Reachability Graphs 62

4 Signal Transition Graphs **68**
4.1 Syntax and Semantics 68
4.1.1 Signal Transition Graphs 69
4.1.2 State Graphs 70
4.1.3 Network functions: Implementations of state graphs. 72
4.1.4 An Example 73
4.2 Obtaining state graphs from STGs 75
4.3 Composition 79

5 Properties of State Graphs **83**

5.1 Liveness 83

5.2 Persistency 84

5.2.1 Definition of Persistency 85

5.2.2 Characterization of Persistency in STGs 89

5.3 A problem with state-assignment 94

6 Decomposition by Net Contraction **97**

6.1 Contraction Algorithms 98

6.1.1 Contraction of Petri nets 98

6.1.2 Contraction of Finite Automata 101

6.2 Properties of Contraction 104

6.3 Decomposition by net contraction 110

6.4 Application to Signal Transition Graphs 118

7 A Design Example **121**

7.1 Specification of the Controller 122

7.1.1 Behavior Specification 123

7.1.2 STG specifications 124

7.2 Synthesis from STG Specification 126

7.2.1 Meeting liveness and persistency 126

7.2.2 Implementation using decomposition 130

7.3 Summary 136

8 STGs with Non-input Choices **138**

8.1 Introduction 138

8.2 The Basic Idea 140

8.2.1 A fundamental problem with specifying non-input choices in state graphs. . . . 141
8.2.2 Specifying non-input choices in state graphs. . . . 142
8.3 STGs with non-input choices . . . 145
8.3.1 Syntax . . . 145
8.3.2 Firing rule . . . 146
8.4 An example: a two-cycle FIFO controller . . . 147

9 The Expansion Algorithm **152**
9.1 Occurrence nets and Processes of nets . . . 153
9.1.1 Occurrence nets . . . 153
9.1.2 Processes of nets . . . 154
9.1.3 A few results for processes of LSFC nets . . . 157
9.1.4 Processes of STGs . . . 158
9.2 The expansion algorithm for STG/NCs . . . 161
9.2.1 Unfolding and folding of free-choice nets . . . 162
9.2.2 The Expansion Algorithm . . . 165
9.2.3 An example . . . 169
9.3 Properties of STG/NCs . . . 172
9.4 Summary . . . 179

10 Suggestions for further research **180**

... All things entail rising and falling timing. You must be able to discern this. In strategy there are various timing considerations. From the outset you must know the applicable timing and the inapplicable timing, and from among the large and small things and the fast and slow timings find the relevant timing, first seeing the distance timing and the background timing. This is the main thing in strategy. It is especially important to know the background timing, otherwise your strategy will become uncertain.

A Book of Five Rings
Miyamoto Musashi

... All Signal Transition Graphs entail rising and falling signal transitions. You must be able to discern this. In our design method, there are various timing considerations. From the initial specification, you must know the allowed timing and the disallowed timing, and from among the large and small circuits and the fast and slow logic gates find the relevant timing, first seeing the distance timing and the timing protocol at the interface. This is the main thing in our design method. It is especially important to know the timing protocol, otherwise your circuits will not work.

Chapter 1

Introduction

1.1 Objectives

The main objective of this research is the development of a design approach for asynchronous self-timed VLSI digital systems. The core of our approach is a graph model called *Signal Transition Graphs* (STGs). STGs allow the specification and efficient synthesis of self-timed control circuits. Our approach produces *speed-independent* logic circuits which can perform *concurrent* operations.

In the realm of VLSI, exploiting concurrency is a prerequisite to high-performance: as systems become larger and more complex, one can no longer afford to ignore the parallelism in control operations. The *central control* discipline which is well-accepted in present approaches creates difficulties in high-performance systems by imposing an unnecessary sequential order on the execution of control operations. In choosing a control discipline which allows for parallel execution of unrelated operations, one naturally moves toward a *distributed control* organization. Thus, instead of a large central controller, one has numerous distributed control modules which can operate concurrently. STGs allow the specification and synthesis of these types of control modules—not only that they can operate concurrently, but also each module itself can perform several control operations in parallel.

By speed-independent circuits, we mean those circuits whose correct operation is independent of the delays of logic gates composing the circuits. One immediate consequence

of this property is that speed-independent circuits are always hazard-free. These types of circuits are desirable in VLSI systems because it is usually difficult to fine-tune the delays of logic gates to make an asynchronous digital circuit work properly. Perhaps most importantly, the use of speed-independent circuits enables the separation of the correctness of systems from timing considerations (which inextricably depend on many physical factors and phenomena in VLSI circuits). It is no coincidence that a number of research efforts in silicon compilation have utilized speed-independent circuits as a basis for hardware implementation [1,42].

STGs allow the direct and efficient synthesis of control circuits from formal specifications. Unlike previous efforts, the approach based on STGs produces a specification which closely reflects the designer's intuition. Moreover, this approach produces efficient circuit realizations by using a number of decomposition techniques. While speed-independent circuits have been criticized for their inefficiency in implementation (and therefore speed and performance), our experiences with applying this graph model to designing VLSI chips have been much more encouraging.

1.2 Background

We can categorize research works in self-timed systems according to two attributes: the theoretical models on which the research is based, and the particular aspects chosen for study. In terms of models, there are finite automata and Petri nets [39,41], and variations of the two. With regard to aspects of study, the two chief areas of concern are composition of systems from modular descriptions, and synthesis of modules from specifications. The study of composition of systems from self-timed modules usually assumes the presence of a set of modules with a certain uniform communication discipline, and the investigation takes a system point of view in exploring properties of systems composed from the modules. The other area of concern concentrates on techniques for synthesis of control modules rather than on the composition and communication aspect of systems.

Two prevalent concepts describing the properties of self-timed systems are ***delay-insensitivity*** and ***speed-independence*** [35,36,37]. While these are commonly used interchangeably, we often find that delay-insensitivity has connoted a stress on the communication aspect and hence, the composition of systems. Thus, this terminology denotes an *external* prop-

erty of control modules. On the other hand, speed-independence is usually understood as a property of control circuits which operate correctly regardless of variation in delays of logic gates. Speed-independence emphasizes the synthesis aspect, hence is a property *internal* to control modules.

Regardless of the emphasis of these research works—whether on composition or on synthesis—they require a formal specification which must rest on the theoretical model chosen. We have found that Petri nets provide a better starting point for formally specifying self-timed circuits, since Petri nets *per se* have capabilities for explicitly modeling concurrency. Also, automata theory originally concentrated on sequential systems. Not until recently was the basic theory extended to cover aspects of concurrency, and not until even later was it applied to the area of logic design.

Below, we briefly review some of the relevant works according to the outline above. In short, each proposal identifies a basic model which also serves as its formal tool for specification, and a focus of study (i.e. composition or synthesis). Interestingly, most of the works deal with only one of the two areas; and none of them provides a coherent framework for treating both synthesis and composition at the same time. This survey is by no means exhaustive; it only serves to highlight the more relevant results which fit more into our classification.

Most techniques for synthesis of asynchronous logic are based on either some form of finite automata such as the Finite State Machine (FSM) model, or Petri nets. The most outstanding work based on finite automata models can be traced back to Muller [35,36,37], who originated the idea of speed-independent circuits. Recent work in trace theory [42,53,51] can be considered as a systematic reformulation of Muller's idea: this refinement is made possible because of the recent attempts to extend automata theory to cover concurrent behavior of systems.

Concerning synthesis approaches based on Petri nets, one of the most important early contributions was made by Patil and Dennis at MIT [15,17,16,38]. Patil invented asynchronous logic arrays as a systematic method of directly implementing Petri nets [38]. In an asynchronous logic array, columns of wires are connected to storage elements to simulate the places of a net, while rows of wires decode the state of the columns to simulate the occurrence of transitions. Thus, this method of implementation transfers the structure of a net directly to hardware. Dennis has shown that Petri nets can be used to model

asynchronous hardware systems at many levels of description in a very clear and easily understood manner [15]. Dennis' description of the control logic of the Control Data 6600 computer, which embodied instruction look-ahead and interleaved execution, demonstrated this technique. Dennis and Patil also elucidated an organization principle for asynchronous hardware systems in which a system is partitioned into data paths and distributed control structures, the latter organized as asynchronous modules which communicate with each other using certain signaling protocols [17,16]. Based on this organization principle, we have successfully designed and fabricated a self-timed two-by-two packet router, a basic component of a packet communication network [10].

Works at Washington University have also made important contributions to the study of self-timed systems, most importantly the use of *macromodules* proposed by Clark [11], and experimental proof of metastability problem in synchronizers, together with design techniques for alleviating this problem [7,13,47]. Most recently, Molnar *et al.* [34] proposed the use of a form of Petri nets called *I-nets* for specifying behavior of control circuits, from which *Interface State Graphs* (ISG) are derived by simulating the I-nets. ISGs can then be encoded with binary states and serve as the basis for implementation of control circuits as standard Huffman asynchronous state machines. This idea was inspired by Seitz's Machine-nets [46] but contained a number of improvements.

Work at CalTech by Seitz resurrected interest in self-timed systems in the VLSI era, as reported in a chapter of Mead and Conway's book *Introduction to VLSI systems* [32]. Currently, Martin [30] at CalTech proposed a design approach using constructs for non-deterministic programming to specify hardware modules whose behaviors exhibit only sequencing and arbitration requirements. This approach uses a subset of Dijkstra's guarded command language [18] to specify each process; concurrently cooperating processes are described using notations similar to Hoare's CSP [25]. Heuristic procedures are used to "compile" a hardware implementation from a module specification into an interconnection of standard hardware templates such as And, Or, C-elements, etc. During the compilation, the technique of reordering signal transitions in a sequence is used to improve implementation efficiency.

Recently, there have been works which use *trace theory* as a formalism for specifying delay-insensitive circuits. Trace theory was pioneered by Mazurkievicz, who has recently made further contributions to this theory [31]. Trace theory has been used in the *COSY*

formalism invented by Campbell and Habermann [6] and in CSP. Rem, Snepscheut and Udding at Eindhoven have demonstrated the use of trace theory for classifying and reasoning about composition of delay-insensitive circuits [42,53,51]. Trace theory has laid a firm theoretical foundation for further investigation of properties of concurrent circuits, as evidenced through recent works [5,45]. Perhaps one novel aspect of these works in comparison with the others is a method for classification of delay-insensitive circuits according to properties of their trace structures [51].

There are a number of earlier works concerning the composition of systems from asynchronous hardware modules; many were reported at the Project MAC Conference on Concurrent Systems and Parallel Computation (ACM, 1970). One notable study was made by Keller at University of Utah, in which he proposed the use of a set of "universal" control modules from which any control network can be constructed [28].

One of the important related works to self-timed systems is the use of temporal logic for verification of asynchronous hardware structures [19]. Such techniques can be used fruitfully for correctness validation of self-timed circuits and systems composed from circuits. It may also be a candidate for a formalism for specification and synthesis of self-timed circuits.

1.3 Problems with some previous models

Below, we describe a number of well-known difficulties with some traditional approaches for designing asynchronous circuits. In particular, we discuss problems with the FSM model and with earlier attempts to apply Petri nets. By identifying these problems, we hope to illustrate the difficult practice of designing asynchronous circuits; this will motivate the search for remedies, some of which are provided by the approach we present in the following chapters.

Traditionally, asynchronous circuits are designed using the FSM model. A method for realization devised by Huffman uses a *flow-table* and a circuit model for implementing the state machine, as described in textbooks such as [20]. This design approach is very difficult to use, especially for synthesizing circuits with many input variables. So far, it has limited applications because of problems caused by variations in gate delays, particular in the

feedback paths of the circuit. Some of the frequently cited disadvantages and limitations of this implementation model are:

- *The Huffman state machine cannot handle unrestricted input changes.* It was discovered by Unger [52] that in the Huffman state machine, if two input transitions occur within a time interval (i) less than $min(D_L + D_f)$, then they can be considered as simultaneous, if (ii) greater than $max(D_L + D_f)$ then they can be considered separate, and if (iii) less that $max(D_L + D_f)$ and greater than $min(D_L + D_f)$, then the secondary state variable will not have settled and the circuit malfunctions; $maxD_L$, $minD_L$ denote the maximum and minimum delays of the combinational logic, and $maxD_f$, $minD_f$ denote those of the feedback delays. (In general, $minD_f$ has to be greater than zero.)
- *The FSM model cannot describe concurrent behaviors directly.* The FSM model and the Huffman state machine are based on the use of central states. At any moment, the machine resides in one state and it reacts to input excitations in different ways depending on which state it is in. A serious drawback of this state-based approach is that it is incapable of describing concurrency directly. The reason is that the notion of concurrency is more conveniently expressed in terms of occurrences of *events*; at the level of description of the FSM model, this phenomenon is difficult to see.
- *The state assignment problem.* Since it is difficult to match gate delays to achieve simultaneous transition of state signals, one has to make sure that simultaneous changes in state signals do not occur, or if they do, the circuit must be designed such that it behaves the same no matter which sequences of state changes take place. Hence most state assignment techniques only allow at most one signal change between states. This is a well-known hard problem for which many heuristic techniques have been proposed. State assignments serve another purpose in the implementation of state machines, that of decomposition. This further complicates the issue, as an optimal state assignment for decomposition may be in conflict with the requirement for single signal changes betweeen states.
- *The exponential dependence of the number of entries in the flow table on the number of input signals.* Due to the absence of a controlling signal called a "clock", a Huffman asynchronous state machine continuously senses the changes in the input and produces changes at the output and the state variables. Therefore, in contrast to a synchronous implementation of a state machine, an asynchronous implementa-

tion requires the listing of all input combinations in the flow table, resulting in the exponential increase in the number of entries in the table.

There are a few early proposals for implementation of asynchronous circuits from Petri net specifications, most notably Patil's asynchronous logic array and Seitz's Machine-nets. Patil's proposal is more correctly viewed as a hardware implementation of Petri nets, rather than implementation of asynchronous circuits from net specifications, because his technique basically replaces an element in a net with a hardware circuit which simulates the behavior of that element. Hence the structure of the net is transferred directly to hardware and there is no direct way to ensure that the resulting asynchronous machine meets all timing requirements. The operation of this type of asynchronous logic arrays depend greatly on local timing, and meeting these timing constraints can sometimes be difficult.

Seitz's Machine-nets, on the other hand, serve only as specifications from which FSM descriptions can be derived. The synthesis of a Huffman state machine from such a description still requires the standard techniques and therefore faces the same difficulties.

1.4 Main contributions of this work

We have developed a synthesis approach for self-timed control circuits from graph-theoretic specifications called *Signal Transition Graphs*, a form of interpreted Petri nets. As will be described in the rest of the thesis, our original contributions are the following.

- A set of new analytical results for Petri nets which allows the study of net properties purely from the syntax (or structure) of nets; we call this the *structure theory* approach.
- The development of Signal Transition Graphs, a formal model for specification and direct synthesis of self-timed control circuits with concurrent deterministic operation and input choices.
- A decomposition technique of nets based on a graph-theoretic notion called *contraction*. This technique can be applied directly to Signal Transition Graphs, thus allowing highly efficient implementations.

- An extension of the Signal Transition Graph model to allow the specification and synthesis of data-dependent control circuits.

1.5 Experimental results

In early 1984, we set out to test our Signal Transition Graph model by designing and fabricating two self-timed chips through MOSIS. The first chip is a two-by-two packet router [10], a basic component of a communication network for a dataflow computer envisioned at MIT [14]. The router was implemented in 3 micron CMOS technology with 2456 transistors and a layout area of $3.1 \times 2.3mm^2$. A total of 46 routers were tested and 30 of them were fully operational with a maximum throughput rate of approximately 22 Mbytes/sec.[1] The other chip fabricated was a self-timed ring buffer, a FIFO buffer with an interesting distributed organization which reduces the latency of the buffer to one stage delay [9]. This ring buffer consists of 8 stages and 9-bit wide data paths. It was fabricated in 4 micron NMOS technology and consumed an area of $3.15 \times 2.25mm^2$, including pads. Six chips were received from MOSIS and tested; five were fully functional at a throughput rate of approximately 4 MBytes/sec. These encouraging results indicate that our proposed approach produces systems which are efficient both in terms of the amount of hardware and speed, perhaps comparable to synchronous implementations.

Description of the Router

The block diagram of the router is shown in Fig. 1.1. It contains two FIFO queues to hold packets sent in byte-serial format. Packets are of variable length, and an extra bit called *Last-byte* is appended to each byte to delimit the packet boundary. This bit is "1" for the last byte of a packet and "0" for all others. The first byte of a packet contains the address information. The router decodes the address and forwards the packet to the desired output port; an address bit of "0" will form a link from the current input port to the upper output port, a "1" will form a link to the lower output port. There are two system controllers, each consisting of a FSM and a *distributed control structure*. The

[1] In [10], the maximum throughput rate for the routers was reported to be 11 Mbytes/sec. This figure is for one input port. The above figure of 22 MBytes/sec reflects the fact that two input ports can process packets concurrently.

system controllers read the address and the *Last-byte* signals, and generate control signals for the output multiplexors. These control signals are determined from the first byte of a packet and recycled for the remaining bytes. The two controllers also communicate with each other, since packets from one input port may need to go to the opposite output port. If packets from both input ports require the same output port at the same time, an arbiter is used to resolve conflicts.

Two main data path modules are the multiplexors, and the self-timed registers which constitute the FIFO's. These modules consist of a data circuit and a stage controller which handles the timing and signaling protocol. These stage controllers are specified using STGs from which speed-independent realizations can be obtained using our synthesis techniques. The controllers contains control circuits which are also synthesized using the same techniques. The *Resource Locking Module* (RLM), a control module with data-dependent operation is treated in an example in Chapter 9. This module is used together with an arbiter [32] to control the access of the FIFO queues to the output multiplexors.

Description of the Ring buffer

The ring buffer is a FIFO queue organized in a two-dimensional or *ring* organization, as shown in Fig. 1.2. This queue consists of M linear queues, each of L stages, and two *token rings* for controlling input/output operation. The capacity of the queue is $M \times L$ and the latency is proportional to L.

Writing into the FIFO queue is controlled by an *Input token ring*, formed by connecting I-modules together into a ring. The ring is initialized such that only one I-module contains the token, marking the next available empty register stage. Since the *Write-request* signal, carried on wire W_r, is connected to all I-modules, an important timing restriction is that the token should not be passed on to the next module in the ring if the *Write-request* signal is still active. The *Write-acknowledge* on wire W_a is the output of an OR gate (shown as a heavy bar with a + sign) whose inputs are *acknowledge* wires from all I-modules. Similarly, reading from the FIFO is controlled by an *Output token ring*, formed by connecting O-modules together. Data written into the linear queues ripple to their output side, ready to be gated onto the output bus. The Output ring is initialized such that only one O-module contains the token. This module then controls the timing and

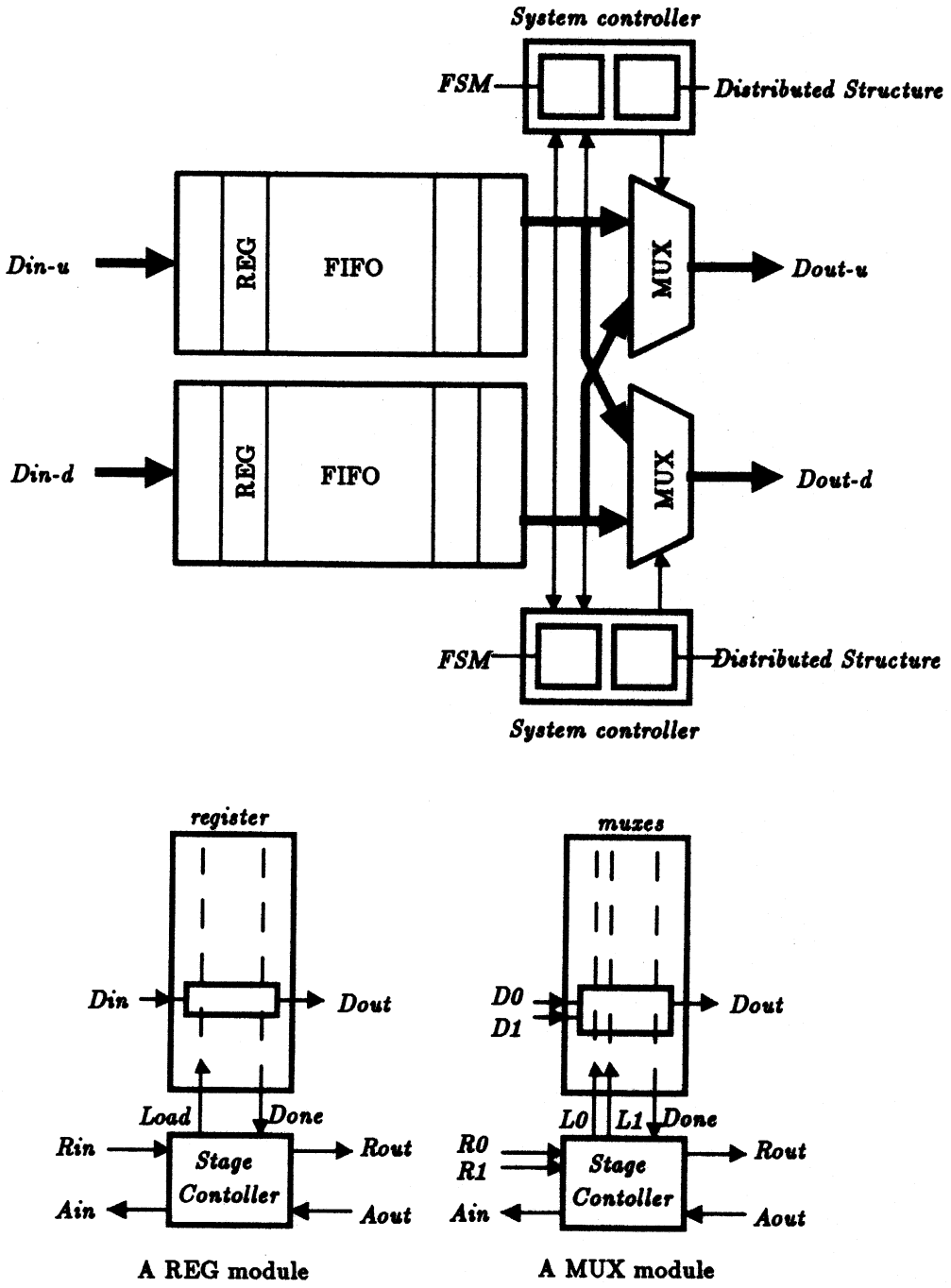


Figure 1.1: Block diagram of the two-by-two packet router.

signaling for gating of data to the output bus. The ***Read-request*** signal on wire R_r is the output of an OR gate whose inputs are ***request*** wires from all O-modules. Another timing restriction exists for the Output token ring: since the ***Read-acknowledge*** signal, carried on wire R_a is broadcast to all O-modules, the token should not be passed on to the next module while ***Read-acknowledge*** is still active.

Our FIFO queue design makes use of distributed control structures and local communication. There are only a few types of modules in this design, with modules of each type replicated as necessary to construct complete FIFO queues. The distributed control structure allows the exploitation of concurrency. Concurrent read/write supports a higher throughput rate. The FIFO queue is also completely data driven, hence no potential read/write conflict exists and there is no need for any arbiter. The distributed control organization of the FIFO lends itself naturally to a design using asynchronous, self-timed hardware modules. These control modules are specified using STGs and synthesized from such specifications. The Ring buffer which we fabricated is one with minimal latency ($L = 1$), with each of the linear queues containing exactly one stage. Registers in each stage have inputs connected to the input data bus, and outputs connected to the output data bus.

1.6 Organization of thesis

After the two introductory Chapters 1 and 2, this thesis is organized into three parts.

Part I consists of Chapters 3–5, in which the basic theory of STGs is developed. This part gives an introduction to Petri net theory, discusses a number of relevant new results and shows how this theory is applied to STGs, a form of interpreted Petri nets. This is followed by an investigation of properties of speed-independent circuits, which translate to the properties of ***liveness*** and ***persistency*** in STGs. The material developed in this part allows the specification and synthesis of a basic type of speed-independent control circuit from STG specifications; these circuits can perform concurrent deterministic operation and (nondeterministic) input choices.

Part II, consisting of Chapters 6 and 7, discusses a novel technique for decomposition of systems based on a notion called contraction. Here, the theoretical results for Petri nets

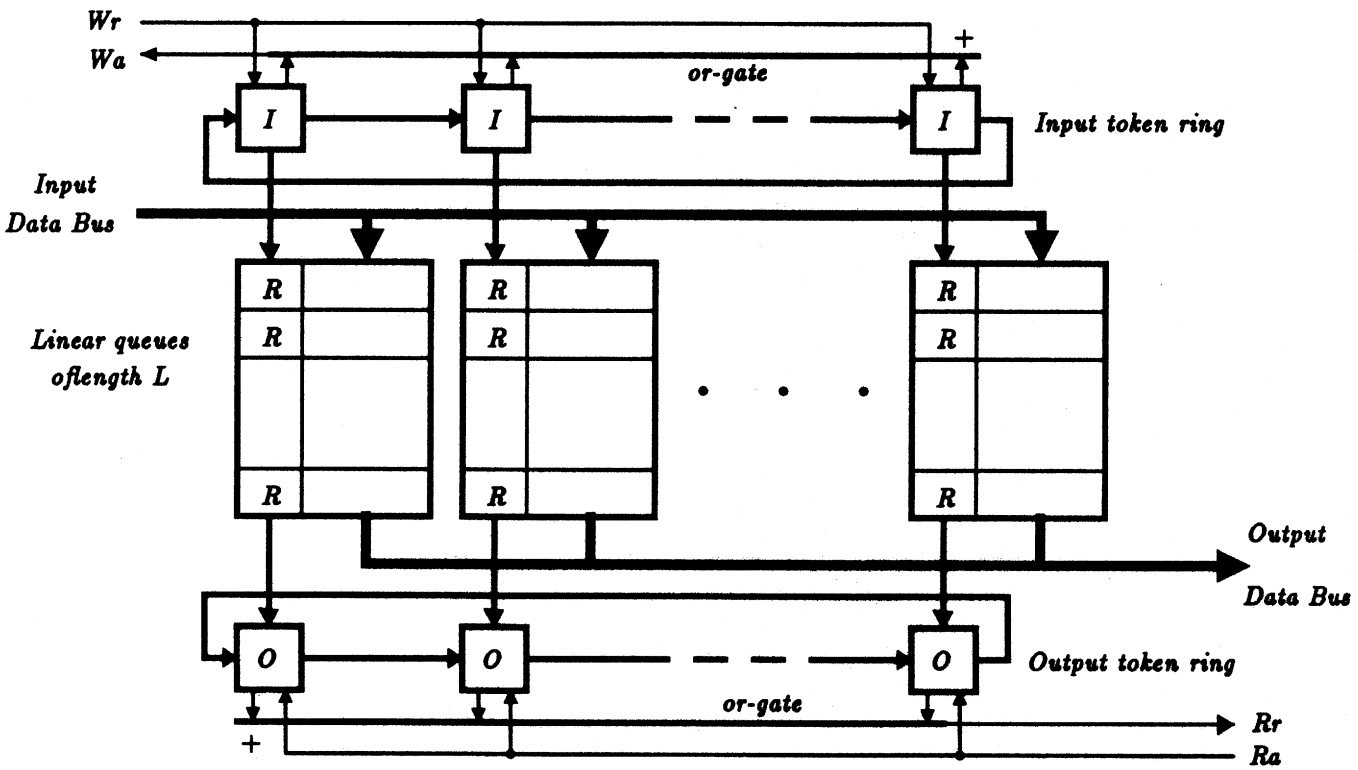


Figure 1.2: Block diagram of the Ring buffer.

are developed and then applied to STGs, as demonstrated through a design example.

Part III, consisting of Chapters 8 and 9, discusses an extension to the basic STG model to allow the specification and synthesis of control circuits with data-dependent operation. The main ideas are presented informally in Chapter 8, together with an example. Chapter 9 develops the formal theory for our extension.

Chapter 2 contains a summary of the STG model and the main theoretical results of Parts I and II. This chapter gives a broad outline of the detailed description which follows in Chapters 3 to 6. It is intended to provide enough knowledge of our methodology for immediate applications. Chapter 7 contains a substantial design example which can be studied without having to go through the technical details presented in Chapters 3–6. Chapter 8 gives an informal presentation of the main idea of our extension of STGs to allow the specification of data-dependent circuits. Thus, after Chapters 2 and 7, Chapter 8 can be read to provide an understanding of the extension.

The following is a more detailed description of each chapter.

Chapter 3. A number of new results are developed for a useful class of Petri nets called live-safe free-choice nets; they constitute the theoretical basis for further investigation of our STG model. First, relevant works in Petri nets theory are reviewed, with particular emphasis on the study of *structure theory*, which is mainly concerned with the relationship between the syntax (structure) and the underlying semantics of nets. This chapter develops two new results: (a) It is demonstrated that the behavior of a free-choice net, as characterized by its set of transition sequences, can be obtained by concurrently composing the behavior of its component subnets. As a consequence, an algorithm is devised, allowing the construction of finite automata directly from the structure of nets by composing finite automata of subnets. (b) A relation called the *temporal relation* on the set of transitions of a net is defined. This relation is characterized based on the structure of nets and it allows for the syntactic determination of whether two transitions are *ordered, concurrent* or *in conflict.*

Chapter 4. The materials developed in Chapter 3 are part of a formal theory based on Petri nets, and they may be useful for other applications as well. For our purposes,

we interpret elements of nets as physical entities of digital circuits: transitions of nets are identified with rising and falling *transitions of signals* in circuits—hence the name *Signal Transition Graphs*. In this chapter, we introduce the STG model, its syntax and semantics, and our design approach for direct synthesis based on STGs. We introduce state graphs and their properties related to physical switching circuits. The third concept in this chapter is *network functions*, defined as the sets of logic functions describing the operation of the circuits. Network functions can be determined directly from state graphs. We also discuss state assignment, being the process of assigning binary values to states in a finite automaton to produce a state graph. Although this is a well-known difficult problem in the classical approach based on the Finite-State Machine (FSM) model, for STGs it is done automatically for STGs by satisfying certain syntactic conditions.

Chapter 5. We discuss two important properties of state graphs and STGs called *liveness* and *persistency*. Liveness is related to the continuous operation without deadlock of circuits; persistency is related to hazard-free operation of circuits. Persistency is strongly tied to the notion of speed-independence: a circuit is speed-independent iff its STG specification is persistent. The equivalent syntactic characterization of liveness and persistency are developed for STGs.

Chapter 6. A method of decomposition for nets called *contraction* is introduced. This allows the decomposition of state graphs through decomposition of their STGs. The purpose of decomposition of state graphs is to produce efficient implementations by minimizing the interaction between variables in the state graphs. While there exist only complex heuristic procedures for FSM approach, for STGs this can be carried easily using net contraction based on some structural information from the STGs.

Chapter 7. This chapter concludes the second part of the thesis by providing a detailed design example of a self-timed controller for a successive-approximation A-to-D converter. We will go through the synthesis steps and illustrate the principles developed earlier. We also discuss a number of design choices available during certain steps of the synthesis process.

Chapter 8. Parts I and II of the thesis provide the ground work which allows one to specify circuit behavior in terms of STGs, obtain their state graphs and finally, produce a correct and efficient implementation by satisfying liveness and persistency. However, since STGs are based on free-choice nets, their expressive power is limited to that of free-choice nets. Since free-choice nets can specify concurrent operations and *free-choices* (nondeterministic choices), correspondingly, STGs can only specify concurrent operations and *input choices* (The reason for considering *free choices* in nets as *input choices* in STGs is that internally to a control circuit module, input choices appear nondeterministic.) In order to specify circuit operation with *internal choices* (data-dependent operation), one will need to rely on a more expressive class of Petri nets.

In Chapter 8, we consider an extension to the STG model which allows the specification and synthesis of circuits with data-dependent operations. This extension marks a slight departure from net theory, as it is a notational extension in STG to represent an aspect of flow-control in state graphs, i.e., the ability to make a decision based on *a priori* knowledge when arriving at a state with conflicts. We discuss this extension and its semantics in terms of state graphs. Lastly, a design example of a two-cycle controller for first-in first-out (FIFO) circuits is presented. This controller is a data-dependent circuit which can perform concurrent control operations.

Chapter 9. Chapter 9 describes an algorithm called *expansion* algorithm which allows the transformation of a STG specifying data-dependent operation into one which has only input choices. Another design example is given: the *Resource Locking Module* which is a part of the controller of the Router discussed earlier.

Chapter 10. We discuss areas for further investigation with the aim toward a comprehensive approach for automatically compiling self-timed VLSI systems from high-level descriptions. Another area of interest is that of optimization of asynchronous circuits based on particular implementation technologies and design methodologies.

Chapter 2

Signal Transition Graphs: An Informal Introduction

This thesis presents an approach for the synthesis of self-timed control circuits from formal graph-theoretic specifications. The conventional approach consists of constructing a Finite State Machine (FSM) from some informal (e.g. textual) description and determining the logic equations for state and output variables from the FSM, as illustrated in the left branch of Fig. 2.1a. Unlike this approach, the basic idea of our approach can be described as follows. From an informal description, we construct a formal specification in terms of graphs called Signal Transition Graphs (STGs), a form of *interpreted Petri nets*. STGs can be considered as a higher level form of representation compared to FSMs in the following sense: from a STG, one can obtain a set of sequences of signal transitions which represent the behavior of a circuit; each transition in every sequence corresponds to a control event of a system. Under certain conditions, such a set of sequences forms a *regular* set, that is, it has an equivalent representation by a finite automaton. This finite automaton can be used as a basis for implementation, as illustrated in the right branch of Fig. 2.1a. Thus, STGs serve as a more abstract and succinct way of representing finite automata with certain desirable properties. STGs are more abstract because these representations use transitions and a binary relation called the *causal relation* between transitions to describe behavior; the concept of state does not appear explicitly. Because of this, concurrency and other control situations can be described in very compact form. STGs are more succinct because they do not require a large number of states to describe concurrent occurrences of control events, in contrast to the case of finite automata.

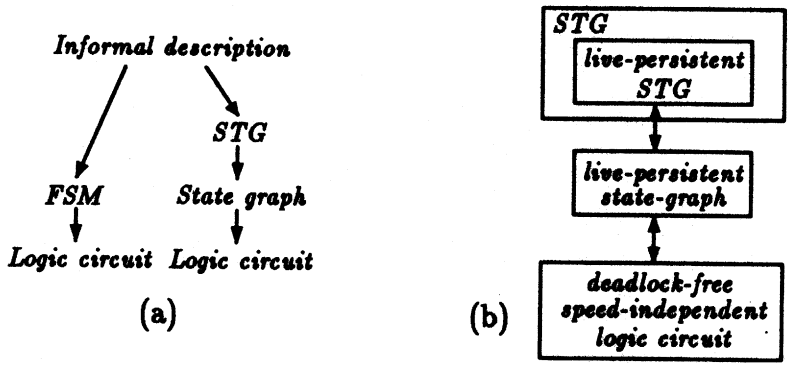


Figure 2.1: (a) Two approaches for synthesis of control circuits. (b) A technique for guaranteeing correctness of implementation from high level specifications.

The finite automata derived from STGs are a form of *interpreted* finite automata called *state graphs*, in which states are interpreted as binary vectors of signal values and transitions between states as signal transitions. State graphs can capture fundamental properties of logic circuits—most importantly the deadlock-free and hazard-free properties. These correspond to the properties of *liveness* and *persistency*, respectively. Since at the higher level of representation of STGs, we do not deal with these low level issues directly, we need to develop some method to ensure correct implementation. Our strategy is as follows: First, we study how fundamental properties of digital circuits can be characterized in terms of state graphs. Then, by establishing a unique correspondence between STGs and their state graph representations, these properties can be formalized as *syntactic* properties of STGs, which can in turns be verified and satisfied at this level of specification. Hence, by developing corresponding syntactic conditions for liveness and persistency for STGs, we have a means for ensuring the correctness of an implementation from an abstract level of specification. This idea is illustrated in Fig. 2.1b.

Another basic notion of our approach is that of *behavioral equivalence*. Some of the previous approaches have used Petri nets not only as a specification but also as a direct basis for implementation, in the sense that the *structure* of a net is directly transferred to hardware. In constrast, we use nets only as a behavioral specification from which a set of transition sequences with certain properties can be derived. Thus nets are considered as language generating devices. Even though this idea has often been studied from the viewpoint of formal language theory, our concern is much more focused and limited to direct and practical applications.

This chapter gives an informal introduction to Petri net theory and summarizes a number of new results, including a simple method for determining the set of transition sequences and its equivalent finite automaton directly from the structure of a net, and a syntactic characterization of the temporal relation between transitions in a net. These results can be applied to STGs as they are merely a form of interpreted Petri nets. The chapter describes properties of state graphs, liveness and persistency, and their equivalent characterization in STGs. Lastly, it presents a method of decomposition based on the notion of contraction; such a decomposition technique is the key to efficient implementation.

2.1 Petri Nets and Signal Transition Graphs

2.1.1 Petri nets

A Petri net is a bipartite directed graph, consisting of a finite set of *transitions* T, a finite set of *places* P and a *flow relation* $F \subseteq P \times T \cup T \times P$ specifying a binary relation between transitions and places. A net is shown in Fig. 2.2a, in which transitions are drawn as bars, places as circles, and the flow relation as directed arcs. One common restriction is that a net be *strongly connected* in the graph-theoretic sense.

Transitions can usually be interpreted as certain *events* in a control system, while places as the local *conditions* which become true or cease to be true due to the occurrence of some actions, as specified by the flow relation. A transition has *input* and *output* places, e.g. p_3, p_4 are input places of t_3; p_5, p_6 its output places. Similarly, a place has *input* and *output* transitions, e.g. t_1 is the only input transition of p_3, t_3 its only output transition. The net in Fig. 2.2a is a particular instance of an important subclass of nets called *marked graphs*, capable of describing systems with deterministic concurrent operation.

A net as presented above describes the static structure of a control system. Its dynamic behavior is captured by its *markings* and the *firing rule* which transforms one marking to another. A marking M is a collection of places corresponding to the local conditions which hold at a particular moment; it is represented graphically as solid circles called *tokens* residing in these places. The *initial marking* is denoted as M_0; in Fig. 2.2a, M_0 corresponds to $\{p_1, p_2\}$. The firing rule is the rule for "executing" a net: A transition is *enabled* if each of its input places contains at least one token. An enabled transition may

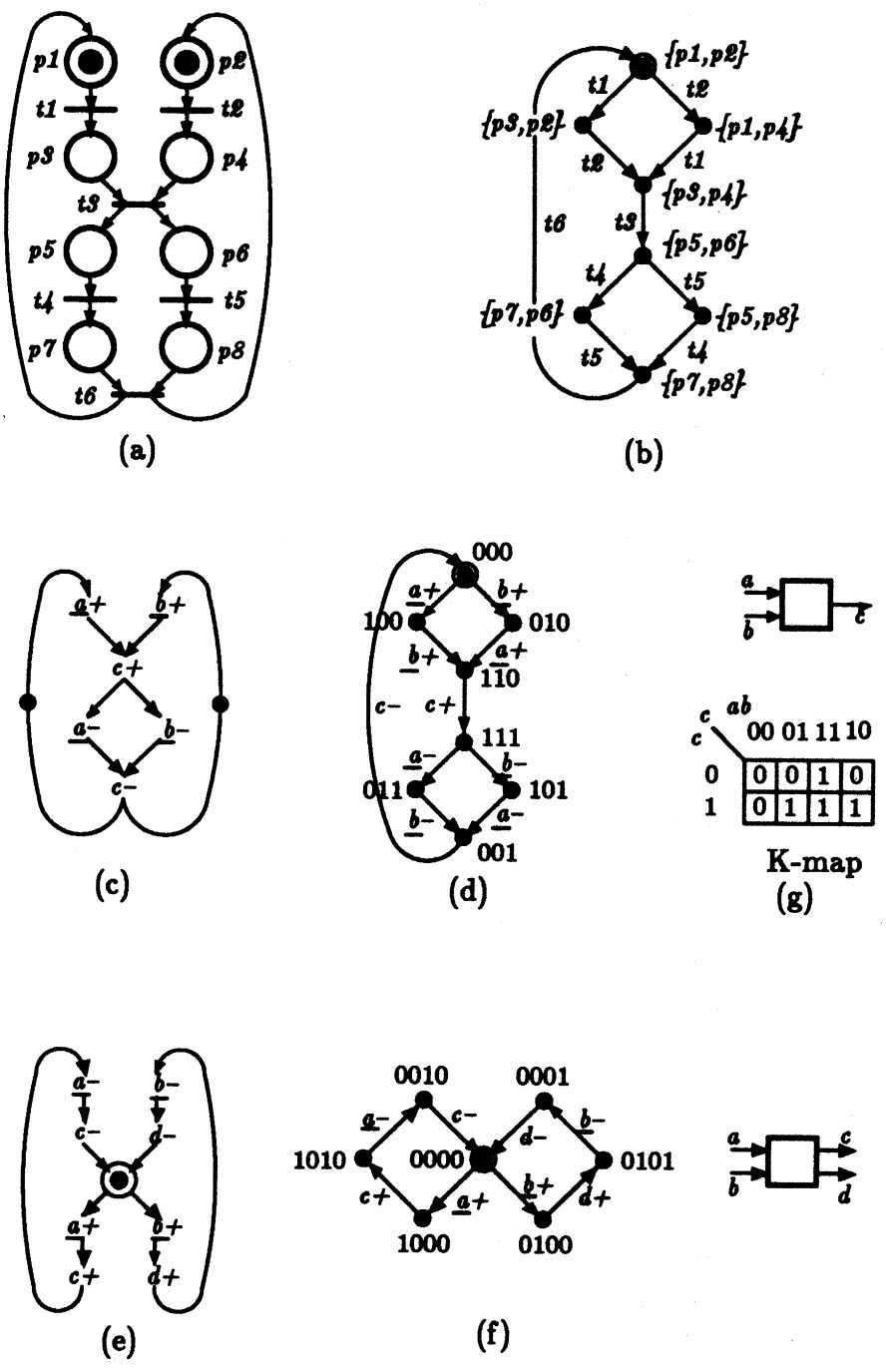


Figure 2.2: (a) A Petri net and (b) its reachability graph. (c) A STG which is an interpreted net of the net in (a), and (d) its state graph. (e) A STG of a circuit with input choices and (f) its state graph.

occur or *fire*; its firing consumes one token from each input place and puts one token in each output place. In Fig. 2.2a, both transitions t_1, t_2 are enabled in the initial marking M_0, the firing of t_1 moves the token from p_1 to p_3, the firing of t_2 moves the token from p_2 to p_4.

The result of the execution of the net can be described by a form of *interpreted* finite automata called *reachability graphs*, as shown in Fig. 2.2b. Each node represents a state corresponding to a marking of the net; a labeled arc between nodes indicates the transition from one marking to another due to the firing of an enabled transition. Also, the *initial state* (corresponding to the initial marking of the net) is circled.

This example illustrates two important points. First, unlike the FSM model, nets can specify *concurrent* control actions: if two transitions are enabled in the same marking and the firing of one does not interfere with the enabling condition of the other, given enough time, *both* transitions will eventually fire. In Fig. 2.2a, the fact that t_1 and t_2 are concurrent means that both transition sequences (or *firing* sequences) $\ldots t_1 t_2 \ldots$ and $\ldots t_2 t_1 \ldots$ are possible; they show up in the reachability graph of the net. Secondly, the net's operation is totally asynchronous, as the firing of transitions depends solely on the availabity of tokens at their input places.

We will stress the transition sequence semantics of nets: *a net defines a set of transition sequences.* For example, the set of sequences specified by the net in Fig. 2.2a can be given by a regular expression

$$((t_1 \| t_2) t_3 (t_4 \| t_5) t_6)^*,$$

where for transitions a and b, $a \| b$ (concurrent composition) denotes the set $\{ab, ba\}$, ab (concatenation) denotes $\{ab\}$ and a^* (Kleene closure) denotes $\{\epsilon, a, aa, \ldots\}$; ϵ is the *empty* sequence.

Properties of nets

Two important behavioral properties of a net with an initial marking are *safeness* and *liveness*, defined as follows. For a net with an initial marking M_0, it is safe iff in any marking reachable from M_0, every place contains no more than one token. For our purpose, only finite safe nets are of practical interest. A finite safe net is live iff its reachability graph is *strongly connected* and each transition in T is enabled in some marking of the reachability

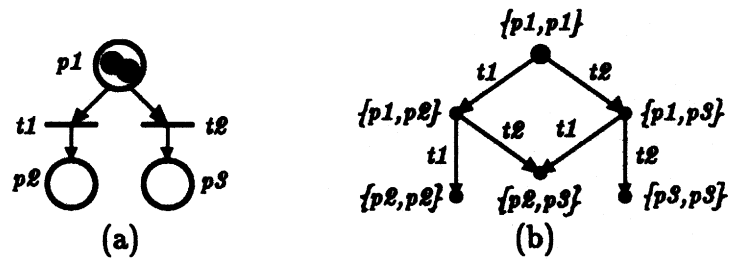


Figure 2.3: (a) An unsafe net and (b) its reachability graph.

graph. Note that this restrictive notion of liveness applies only to finite safe nets, and further it requires that all markings be reproducible.[1] The purpose of this requirement is to permit one to disregard the transient behavior during initialization of nets.

We consider safeness as a fundamental restriction on nets; without it one cannot relate the structure of a net to the actual behavior which the net intends to describe. Fig. 2.3a is a simple example of an unsafe net, whose structure is intended to specify a choice between control actions t_1 and t_2. This would have been the case had place p_1 contained only one token. However, due to the fact that place p_1 contains two tokens, t_1 and t_2 each can fire twice consecutively, or both of them can fire concurrently. This behavior is recorded in the reachability graph shown in Fig.2.3b, with each marking described by a multiset of marked places (instead of a set) due to the multiplicity of tokens in the places.

Unsafe nets create another fundamental problem in that the set of transition sequences derived from an unsafe net may not have an equivalent finite automata (FA) representation in case the number of tokens in any place grows without bound.

Subclasses of nets and Structure theory

We will be concerned with three important subclasses of nets called *marked graphs* (MG), *state machines* (SM) and *free-choice* (FC) nets. A marked graph is a net in which each place has at most one input transition and at most one output transition. Marked graphs represent the structure of deterministic concurrent systems. The dual notion of marked graphs is that of state machines. A state machine is a net in which each transition has at most one input place and at most one output place. State machines represent the

[1] In Fig. 2.4a, the initial marking $\{p_1, p_4\}$ is a live-safe one; however, this marking is not reproducible.

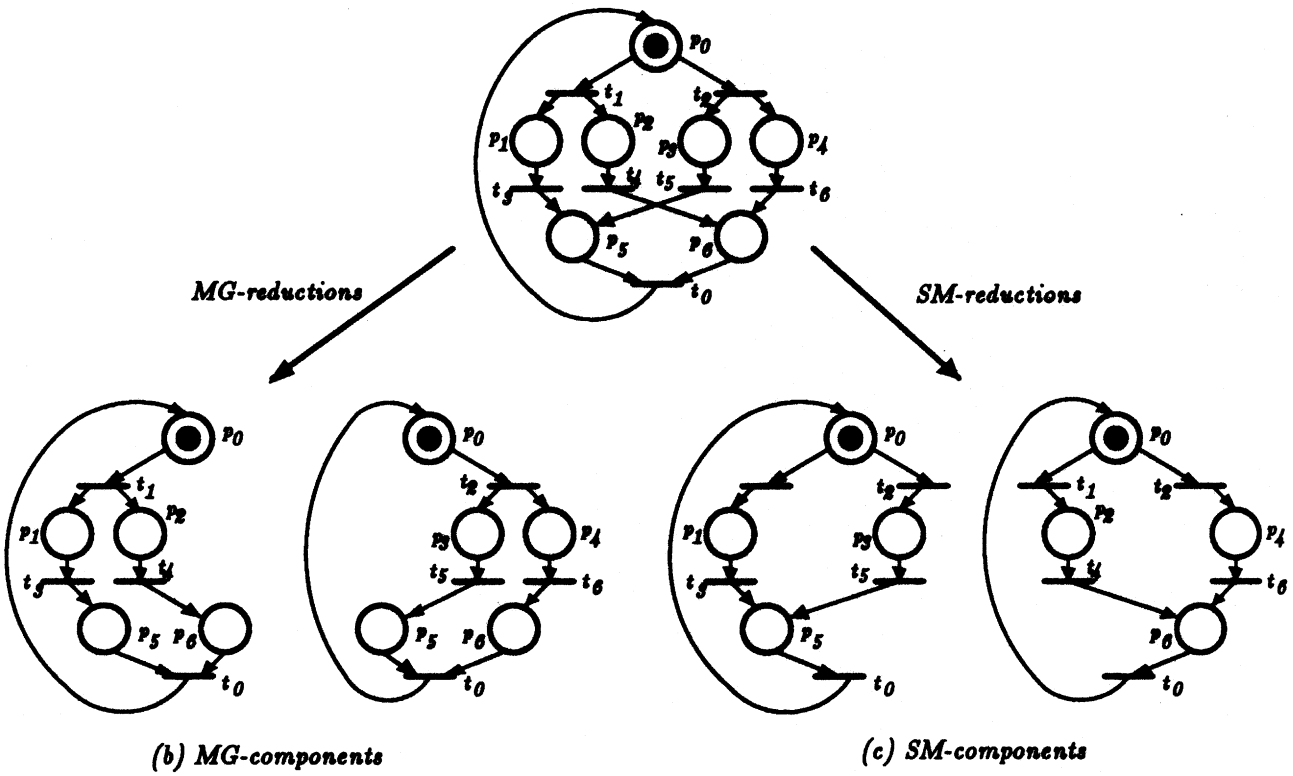


Figure 2.4: (a) A LSFC net, (b) its MG-components resulting from MG-reductions and (c) its SM-components resulting from SM-reductions.

structure of nondeterministic sequential systems. A free-choice net is a net such that if any two transitions t_1 and t_2 share the same input place p, then p is the unique input place of both t_1 and t_2. Examples are shown in Fig. 2.4, where (a) is a FC net, (b) consists of marked graphs and (c) of state machines. We will restrict ourselves to a subclass of FC Petri nets as specifications of control systems which exhibit concurrent operations. FC nets represent an appropriate blend of concurrency and choice in specifying behaviors of circuits, and at the same time can be analyzed without much difficulty.

Structure theory is a branch of net theory which emphasizes the relationship between the structure (syntax) of nets and their behavior (semantics). In our view, structure theory is vital to the practical use of nets, for it allows the characterization of dynamic properties of nets in terms of static (syntactic) ones. Hack [24] has devised a *reduction* algorithm which allows the decomposition of a free-choice net into sets of structural components: a FC net can be decomposed into a set of state-machine (SM) components or a set of

marked graph (MG) components, as shown in Fig. 2.4. An important theorem developed by Hack which is the cornerstone of the structure theory of Petri nets can be informally stated as follows. If a FC net is live and safe, then the set of MG-components resulting from MG-reductions *covers* the net. Alternatively, the set of SM-components resulting from SM-reductions also covers the net. On the other hand, if a FC net is either nonlive or unsafe, then some reduction does not cover the net or is empty, or some component is not strongly connected. For example, the FC net in Fig.2.4 is live-safe and its SM-and MG-components both cover the net.

Hence, by using this theorem, we can determine if a net is live-safe by decomposing it into structural components. Later chapters describe more fully other important applications of this theorem. Specifically, by using Hack's theorem, we developed techniques for constructing a finite automaton directly from the structure of a net, and a syntactic characterization of the temporal relation, as will be described below.

2.1.2 Signal Transition Graphs

For the purpose of specifying behaviors of digital control circuits, we use a form of *interpreted* Petri nets called *Signal Transition Graphs* (STGs), which are nets with transitions interpreted as rising and falling transitions of signals of a control circuit. Fig. 2.4c shows an example of a STG of a circuit with the set of signals denoted by $J = \{a, b, c\}$. This STG is one interpretation of the net in Fig. 2.2a, where the set of transitions T is interpreted as the set of *signal transitions* $J \times \{+, -\}$. Since a control circuit has input, internal and output signals, we partition the set of signal transitions in the same way, and in the graphical representation, transitions of input signals are underlined. The fundamental difference between transitions of input and *non-input* (= internal + output) signals is that the former are caused by the external environment while the latter by the system.

For simplicity, in a STG, we represent each transition by its name instead of using a bar with a label. Another important graphical abbreviation for STGs is that every place with one input and one output transition is not drawn explicitly; instead an arc is drawn directly between these transitions. Such an arc directly represents an instance of the *causal* relation, denoted by R, between transitions; informally, $t_1 R t_2$ (read t_1 *causes* t_2) can be understood as: the firing of t_1 brings the system into a state (marking) in which t_2 is

enabled (and hence may fire).

The reason for using the causal relation and ignoring places with one input and one output transitions is that, from the viewpoint of firing sequence semantics, the behavior of a net (or STG) is adequately defined by its set of transition sequences. For a sufficiently expressive class of nets called *live-safe free-choice* (LSFC) nets, we will be able to show that their sets of firing sequences are *regular* in the sense that the latter have equivalent finite automata representations. For example, the STG in Fig. 2.2c defines a set of sequences which has the equivalent finite automaton (FA) shown in Fig. 2.2d, which is isomorphic to the reachability graph of Fig. 2.2b. This FA can be interpreted into a *state graph* by (i) identifying transitions between states with signal transitions and (ii) assigning binary vectors representing the values of signals in the circuit to nodes. For state graphs, this state assignment is very simple and can be carried out mechanically, as will be described later.

The STG in Fig. 2.2c specifies the behavior of a control circuit with deterministic concurrent operation. Fig. 2.2e shows another STG specification of a trivial circuit with *input choices*; its state graph is shown in Fig. 2.2f. The input choice is specified by a place with two output transitions a_+ and b_+, which are transitions of input signals a and b. Whenever p is marked, then both output transitions are enabled and one is chosen nondeterministically to fire; its firing will *disable* the other transition. In Petri nets, such a situation is called a *free choice*. In STGs, we limit output transitions of a free-choice place to those of input signals because internal to a system, a choice made externally would appear as if it is nondeterministic.

We have the following important remarks concerning the formalization of STGs as interpreted Petri nets.

First, STGs are nets with interpreted transitions but places have no interpretation; in particular, we do not interpret places as states of signals resulting from the firing of signal transitions. The reason is illustrated in the following example. In Fig. 2.2a, transition t_1 is interpreted as a_+. Suppose further that its output place p_3 is interpreted as signal a becoming a logical "1". Then whenever t_1 fires, indicating a positive transition of a, place p_3 is marked with a token indicating that signal a has become 1. Note, however, than when t_3 fires subsequently, this token is taken away, implying that signal a is no longer at value 1. In reality this is not true because the value of a will not change until a_- occurs.

This is an important observation to which we shall return later.

Second, despite the fact that STGs are interpreted nets, we have chosen to call our graph model Signal Transition *Graphs*, rather than *Nets*, in order to emphasize the important role of the static structure of nets in our application. A Petri net can be viewed as consisting of an underlying structure which is a graph, and a marking which indicates the distribution of tokens in the graph at some moment. Structure theory, as mentioned earlier, allows the direct association of static net structures to their underlying semantics. This thesis develops a number of techniques for manipulating the structure of nets (with little concerns about markings) which allow direct synthesis of the underlying finite automata.

2.1.3 Relation between Petri nets and Signal Transition Graphs

It is important to note that STGs can be considered simply as a class of LSFC nets with certain structural restrictions due to the interpretation of transitions. One such restriction is that STGs always contain an even number of transitions due to the fact that associated with every signal is a pair of signal transitions. Another restriction arises from the need to simulate the interface behavior of a control circuit: in a STG, if t is a transition of an input signal, then we require that t have exactly one transition, say t', which causes it: $t'Rt$; furthermore, t' must be a transition of an output signal. In Fig. 2.2c, each transition of input signals a, b is caused by exactly one transition of output signal c. There are other restrictions which we will present later.

Just as a STG is an *interpreted* net, its state graph is an *interpreted* FA which can be obtained in a similar fashion as reachability graphs. The relationship between nets, STGs and state graphs is depicted in Fig. 2.5. The top part of this figure is a *syntactic* or *structural* classification of nets. For instance, the class of FC nets is a subset of the class of Petri nets, and so on. LSFC nets are a subclass of FC nets, with the properties of live-safeness characterized structurally by Hack's reduction theorem mentioned at the end of Section 2.1.1. STGs constitute a subclass of LSFC nets. The subset of *live-persistent* STGs corresponds to those STGs whose state graphs are live and persistent. Live and persistent state graphs can be transformed into deadlock-free and hazard-free (speed-independent) logic circuits, as indicated by arrow (5).

The middle part of Fig. 2.5 contains classes of finite automata, which can be considered

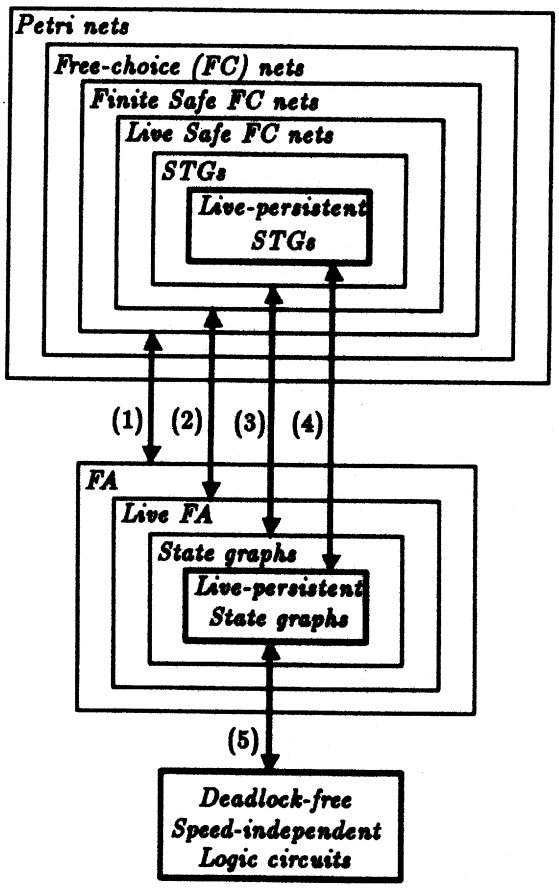


Figure 2.5: Structural classification of classes of nets and their equivalent finite automata.

as the low-level semantics of nets. The two-headed arrows indicate the equivalence between them and classes of nets. Earlier, we stated that safe nets have equivalent FA representations; such an equivalence is indicated by arrow (1) for finite *safe* FC nets. The class of LSFC nets have equivalent FA which are live, as indicated by arrow (2), this equivalence is the subject of investigation of Section 2. Of main interest in Fig. 2.5 is the equivalence indicated by arrow (4) between live-persistent STGs and state graphs; this equivalence is studied in Section 3.

It is crucial to realize that while safeness can be considered purely as a net-syntactic property, liveness and persistency are defined as properties of finite automata. Thus, live-safe FC nets are safe FC nets whose FA are live; similarly, live-persistent STGs are STGs whose state graphs are live and persistent. The important point is that even though liveness and persistency are defined as properties of state graphs, one can derive the equivalent syntactic conditions for STGs, just as in the case of LSFC nets. For LSFC nets, the syntactic conditions for live and safeness are stated in Hack's decomposition theorem.

2.2 Semantics of nets/Behavioral equivalence

There are two approaches to defining the semantics of Petri nets: one based on sequences of transitions (firing sequences), the other on partial orders of transitions and places. For safe nets, their semantics can be given both in terms of firing sequences and partial orders: By using the firing rule and an initial marking, one can simulate the operation of a Petri net to obtain the firing sequences; on the other hand, Petri nets can be unfolded into partial orders called *processes* [23].

We will be mainly interested in the firing sequence semantics of Petri nets. One frequently cited problem with using firing sequence semantics is illustrated in the following: Given a set of firing sequences describing the behavior of a very simple system $\{ab, ba\}$ (Fig. 2.6a), it is unclear whether this set corresponds to a net in which a and b are concurrent (Fig. 2.6b) or in conflict (Fig. 2.6c), where all transitions are labeled.

It is clear that this problem arises due to the labeling of two transitions with the same labels in Fig. 2.6c. If such a labeling is disallowed, problems of this kind will never arise and the system $\{ab, ba\}$ always corresponds to the case of *concurrency*. Hence we allow

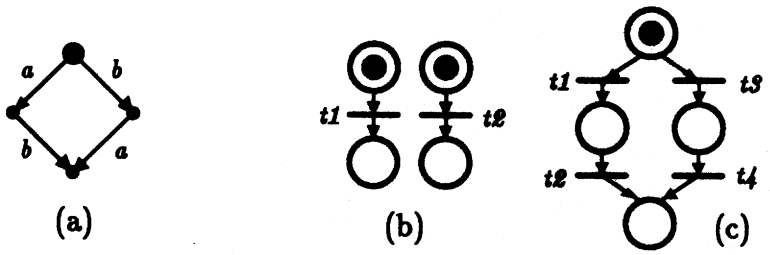


Figure 2.6: (a) The equivalent FA of the set $\{ab, ba\}$ and two interpretations: (b) concurrency, where $\gamma(t_1) = a$, $\gamma(t_2) = b$; γ is a ***labeling*** function, (c) conflict, where $\gamma(t_1) = \gamma(t_4) = a$, $\gamma(t_2) = \gamma(t_3) = b$.

only *unlabeled transition sequences*. In the formulation of STGs, instead of using a labeling function to label transitions of a net with signal transitions, i.e. $\gamma : T \rightarrow J \times \{+, -\}$, we have chosen to make a direct interpretation: $T = J \times \{+, -\}$ to avoid the problem just mentioned.

2.2.1 Behavioral Equivalence

For our purpose of synthesis from net specifications, the main advantage of using sequence semantics is that it allows freedom in the implementation from a specification. Given a net specification of a control systems, any other net which exhibits the same set of transition sequences as the original one is considered ***equivalent*** to it; this is the notion of ***behavioral*** equivalence. The like notion of "structural equivalence" is not a particular useful one, as it requires two nets which are equivalent in this sense to have identical structures. The simple example in Fig. 2.7 shows two nets which are behavioral equivalent but not so structurally. If one is concerned only about the behavior (in terms of transition sequences), then the net in Fig. 2.7a contains a redundant place which can be removed without altering the net's behavior.

From a technical point of view, sequence semantics are easier to handle than semantics based on processes. As will be made clear in this thesis, most results for LSFC nets are proven by considering their sets of transition sequences. These include important properties such as liveness, and properties concerning the composition and decomposition of nets.

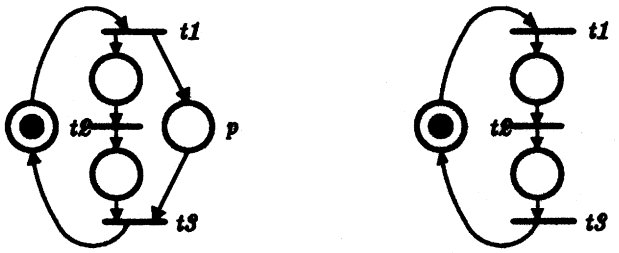


Figure 2.7: Two nets which are behaviorally equivalent.

2.2.2 Important Analytical Results for Nets

In this section, we summarize a number of fundamental results which are proven in Chapter 3. These results serve as the basis for later developments. One important result is the following

> *The set of firing sequences of a LSFC net is regular, i.e. it has an equivalent finite automata representation.*

Algorithm for constructing finite automata for LSFC nets

The above result is proven by demonstrating a procedure for constructing an equivalent automaton from a LSFC net; the basis for such a construction algorithm is given by the following result (Theorem 3.8):

> *The equivalent finite automaton of a LSFC net is obtained by weaving (concurrent-composing) the finite automata of a set of SM-components which cover the net.*

Note that a FA equivalent to a state machine is identical to the state machine itself and hence, can be readily derived from the SM-components of a LSFC net. Let FS_1 and FS_2 denote sets of transition sequences of two finite automata FA_1 and FA_2, respectively. The weave of FS_1 and FS_2 is defined as (Def. 3.7)

$$FS_1 \| FS_2 = \{\sigma \in (T_1 \cup T_2)^* \mid \sigma\lceil T_1 \in FS_1 \ \wedge \ \sigma\lceil T_2 \in FS_2\},$$

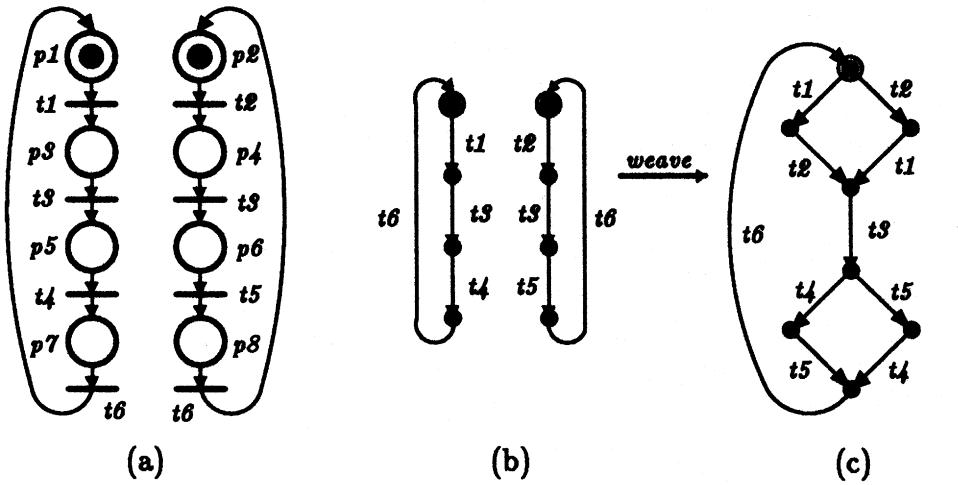


Figure 2.8: (a) The set of covering SM-components for the net in Fig. 2.2a. (b) The equivalent FA of the SM-components and (c) their weave, which results in the equivalent FA of the net in Fig. 2.2a.

where T_1 and T_2 are sets of transitions of the FA; $\sigma\lceil T_i$ denotes the *projection* of a sequence σ onto the set T_i. The weave of two FA, $FA_1 \| FA_2$, is defined as a FA whose set of transition sequences is given by $FS_1 \| FS_2$.

The marked graph in Fig. 2.2a is a special case of LSFC nets, which can be decomposed into a set of covering SM-components (Fig. 2.8a), each being a simple cycle containing one token. The FA corresponding to these SM-components are shown in Fig. 2.8b; their weave results in the FA for the marked graph, as indicated in Fig. 2.8c.

Temporal relation in LSFC nets

Another result of fundamental importance developed in this thesis could be categorized as a result in the structure theory of Petri nets. Given a LSFC net, it is generally not possible to tell whether two transitions are ordered or concurrent merely by inspecting the structure of the net. For instance, the marked graph shown in Fig. 2.9 (called a *necklace* of length 4) has three different initial markings as indicated, each resulting in a distinct set of firing sequences; that is, this net has three equivalent classes of live-safe markings—every two markings in each class are mutually reachable. In case (a), the set of firing sequences is given by the expression $(t_1t_2t_3t_4)^*$; in case (b), it is $(t_1t_4t_3t_2)^*$; lastly in case (c), it is

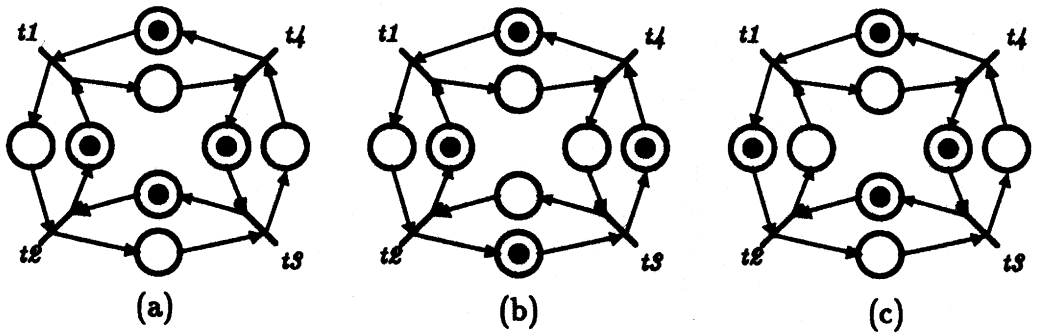


Figure 2.9: (a) A FC net with three different equivalent classes of live-safe markings. Some of its SM-components contain two or three tokens.

$t_2((t_1 \| t_3)(t_2 \| t_4))^*$. In cases (a) and (b) all transitions are ordered, even though they are ordered in different ways. In case (c), t_1 and t_3 are concurrent, and so are t_2 and t_4. For each of these cases, there exists a set of covering SM-components which are simple cycles with one token. Note, however, that there are also cycles with two or three tokens. Since by definition a state machine net must contains exactly one token, such a cycle with two tokens is not a legitimate SM-component.

In order to allow for syntactic characterization of the temporal relation between transitions in a net, we shall make the following fundamental restriction. The original result by Hack, as described in his Well-formedness theorem, states that a FC net is live-safe iff when SM-reductions are applied to it (i) every SM-reduction is a collection of one or more marked SM-components and (ii) the reductions *cover* the net. Note that this condition does not require that *every* SM-component of the net contains one token each; it is possible for *some* SM-components to contain more than one token, as illustrated by the example above.

Our temporal characterization based on syntax only works for nets which satisfy the *one-token SM restriction*:

Every *SM-component of a LSFC net contains exactly one token.*

Hence, if net satisfies this restriction, *any of its SM-components containing more than one token is unsafe and will cause the net to be unsafe.* In this thesis, we will be mainly interested in LSFC nets which meet this restriction. This is not as restrictive as it seems:

First, any LSFC nets not satisfying this restriction can usually be decomposed into subnets, each of which satifies it. Secondly, many other models of concurrent systems including *CSP* [25], *path expressions* [6], *etc.* require that every basic module be a sequential process—concurrency is achieved by having many sequential modules communicating with each other. In contrast, our one-token SM restriction does not require each basic module to be a sequential process. The advantages for introducing this restriction are:

- It allows a simple and useful characterization of the temporal relation between transitions based solely on the structure of the net. One example of this characterization is that two transitions are ordered iff they belong to the same simple cycle in a LSFC net. Such a characterization is correct only when every simple cycle contains exactly one token.
- Each net in this class of nets has exactly one equivalence class of live-safe markings and furthermore, the equivalence class can be determined directly from the structure of the net. This has profound implications since in general, a LSFC net may have more than one equivalence class of live-safe markings if some of its SM-components contain more than one token (an example has been given in Fig. 2.9). The problem of determining all different classes of live-safe markings is a major difficulty in net theory [Hol74]. Moreover, if a net has only one equivalence class of markings, it is possible to determine all markings in this class directly from the structure of the net. Therefore, the concept of an initial marking serves merely as an indicator of the starting state of a net and plays no significant role in the synthesis from net specifications.

Under the one token SM restriction to LSFC nets, we can define a relation called the *temporal relation* $\mathbf{tr} = T \times T$. The temporal relation is a binary relation on the set of transitions (and can be extended to include places) of a LSFC net, and is defined based on the structure of nets. It allows one to determine syntactically whether two transitions are *ordered, concurrent* or *in conflict.*

In the following, we give the net-syntactic definition of symmetric binary relations **li**, **co**, **cf** and **dc** on the set of transitions; they stand for *ordered, concurrent, conflict* and *direct-conflict*, respectively. In a LSFC net, two transitions t and t' are (a) *ordered*, denoted as $\{t, t'\} \in \mathbf{li}$ iff there exists a simple cycle containing both of them; (b) *concurrent*,

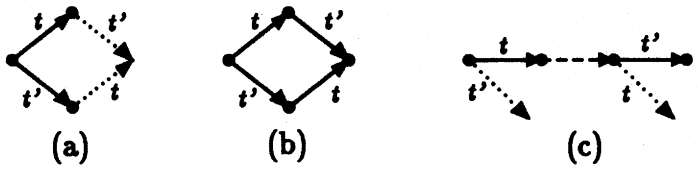


Figure 2.10: Portion of a reachability graph showing t and t' are (a) in direct-conflict (b) concurrent (c) ordered or in *indirect*-conflict.

denoted as $\{t, t'\} \in \mathbf{co}$, iff there exists a MG-component containing both of them, but not in the same simple cycle; (c) otherwise, they are *in conflict*, denoted as $\{t, t'\} \in \mathbf{cf}$. One special case of conflicts is a *direct conflict*, denoted by $\{t, t'\} \in \mathbf{dc}$, iff there exists a place of which t and t' are output transitions. For example, in Fig. 2.4, $\{t_1, t_4\} \in \mathbf{li}$, $\{t_5, t_6\} \in \mathbf{co}$, $\{t_2, t_3\} \in \mathbf{cf}$ and $\{t_1, t_2\} \in \mathbf{dc}$.

The above classifications of the temporal relation are based on the syntax of LSFC nets. It can be proven that these correspond to the normal understanding of ordering, concurrency and conflict in terms of firing sequences. For a LSFC net, there exists a correspondence between an element of the temporal relation and a situation in a reachability graph of the net, as illustrated in Fig. 2.10. In this figure, dotted arcs are used to indicate explicitly those transitions which cannot occur; dashed arcs indicate directed paths between markings.

Another result concerning this syntactic characterization is the following. In a LSFC net satisfying the one-token SM restriction, if we define the *temporal relation* on the set of transitions as $\mathbf{tr} = T \times T$, then it can be shown that (Theorem 3.16):

> **li, co** *and* **cf** *partition the temporal relation* **tr** *into disjoint subsets, and* $\mathbf{tr} = \mathbf{li} \cup \mathbf{co} \cup \mathbf{cf}$.

The above syntactic characterization plays an effective and complementary role to the construction algorithm developed earlier: In a net, while the temporal relation allows the *syntactic* characterization of a behavior specification (i.e. whether certain control events are concurrent or in conflict, etc.), the finite automaton obtained from the net provides the basis for further analysis and synthesis. Hence, we have introduced enough analytical results to permit the use of nets as formal specifications of control systems, which can be

viewed as high level representations of finite automata for the purpose of implementation.

2.3 Properties of State graphs

In this section, we briefly discuss interpreted finite automata called *state graphs* and their logic implementations, as described by *network functions*. A network function is simply a set of logic equations for non-input signals in a circuit, and it can be obtained directly from a state graph.

We will further explore two important properties of state graphs called *liveness* and *persistency* and derive corresponding conditions on STGs. As mentioned earlier, these will appear as syntactic conditions on STGs, and by satisfying them, correct hardware implementation is guaranteed: simply, a control circuit is deadlock-free and hazard-free if its STG specification is live and persistent.

2.3.1 State Graphs and Network Functions

In the construction of an equivalent FA from a set of sequences, a state is an abstract concept, defined as an equivalence class of sequences with the same postfixes; there is no apparent relationship between a state and the transitions enabled in it. However, for state graphs obtained from a STG, not only do we require that it be a FA equivalent to the set of sequences defined by a STG, but we also require that *states be interpreted as binary vectors representing the values of signals in a circuit.* There is a direct connection between states and transitions: states are vectors of values of a set of signals, whereas transitions are transitions of the same set of signals. For example, in Fig. 2.2d, the control circuit is comprised of the set of signals $J = \{a, b, c\}$. Hence in its state graph, every state is interpreted as a binary vector representing the values of signals in the ordered set $\langle a, b, c\rangle$ and every transition is a signal transition in $J \times \{+, -\}$. Such a connection requires that every state s be assigned values in a manner consistent with transitions from s. For a signal $j \in J$, let $s(j)$ denote the binary value of signal j in state s, and j_* denote a transition of j (either j_+ or j_-). Also let $s[t\rangle s'$ assert that the occurrence of transition t in a state s takes the system to another state s'; t is said to be enabled in state s. In a state graph, if $s[t\rangle s'$ where $t = j_*$, then s, s' and t must together satisfy the following condition: if $t = j_+$

then $s(j) = 0, s'(j) = 1$; if $t = j_-$ then $s(j) = 1, s'(j) = 0$. In which case, the triple $\langle s, t, s' \rangle$ is called *consistent*, and s, s' are *adjacent*. For example, in Fig. 2.2d, $s = 000$, $t = b_+$ and $s' = 010$.

A state graph is said to have a *consistent state assignment* iff the above condition holds for all states in the state graph. A state graph has a corresponding logic implementation described by a network function which can be determined from the state graph. A network function consists of a set of logic equations, one for each non-input signals to be implemented.

In a state graph, the logic equation for every non-input signal j can be determined as follows. For states s, s' and transition t such that $s[t\rangle s'$, s' is called a *next-state* of s. For a signal j, its *implied value* in state s, denoted by $f(s, j)$, is determined by: if t is a transition of signal j, i.e. $t = j_*$, then $f(s, j) = \bar{s}(j)$; otherwise $f(s, j) = s(j)$, where $\bar{s}(j)$ denotes the complement of $s(j)$. Notice that in a state graph, it is often the case that a state s has more than one next-states s'; however, each state can have exactly one implied value. The logic equation for j is determined from the set of implied values of j in all states. This can be done conveniently by transferring the state graph to a Karnaugh map (K-map) such that in a square of the K-map corresponding to some state s, the implied value of s is written. For example, to determine the logic equation for signal c, the state graph in Fig. 2.2d can be transferred to a K-map; and it can be verified easily that $c = a.b + c(a + b)$. This is illustrated in Fig. 2.2g, and the logic function is precisely that of a C-element.

2.3.2 Liveness and Consistent State Assignment

In the process of obtaining a state graph from an STG, we first determine the equivalent FA of the STG and then perform state assignment for the FA. Performing state assignment is essentially interpreting the FA. In the last section, the concept of consistent state assignments has been described. It follows that in every simple cycle of a state graph with consistent state assignment, the numbers of t and $\bar{t}$ transitions must be equal, and $t, \bar{t}$ must alternate; where $t, \bar{t}$ denote a pair of complementary signal transitions. We have derived the following corresponding condition on a STG for its state graph to have a consistent state assignment (Theorem 4.8).

A state graph of a STG has a consistent state assignment iff every pair of transitions $t, \bar{t}$ is ordered.

The continual operation without deadlocking of control systems is the property of *liveness*. A state graph of a control system is called *live* iff its uninterpreted FA is strongly connected and has a consistent state assignment. If the state graph is strongly connected then all of its transitions occur infinitely often in some transition sequences.

A STG is said to be *live* iff its state graph is live. The syntactic conditions on a STG so that its state graph is live are the following. (Theorem 5.1)

A STG is live iff (i) its uninterpreted net is a LSFC net, and (ii) every pair of transitions $t, \bar{t}$ is ordered.

Condition (i) guarantees that the FA is strongly connected; condition (ii) guarantees that the interpreted FA (i.e. the state graph) has a consistent state assignment. Recall that a strongly connected FA is said to be live. An example of a live STG and its state graph are given in Fig. 2.2 above.

2.3.3 Persistency

Persistency is one of the most important properties of state graphs. Persistency is an important concept because it is the essential property of speed-independent circuits. It is also the most complicated to deal with because there may be a number of mechanisms involved.

In a state graph, a transition is *persistent* iff when it is enabled, the occurrence of some other transition does not disable it. This situation is illustrated in Fig. 2.11a: transitions t and u are enabled in state s and $s[t\rangle s_1$, $s[u\rangle s_2$. t is persistent, as the occurrence of u does not disable it. On the other hand, u is *non-persistent* because it is disabled by an occurrence of t and hence not enabled in s_1. The disabling of u must have been caused by the occurrence of transition t, as t is the only transition which occurs in going from state s to s_1. In which case, t must be a transition of an input to a logic element of which u is a transition of its output signal.

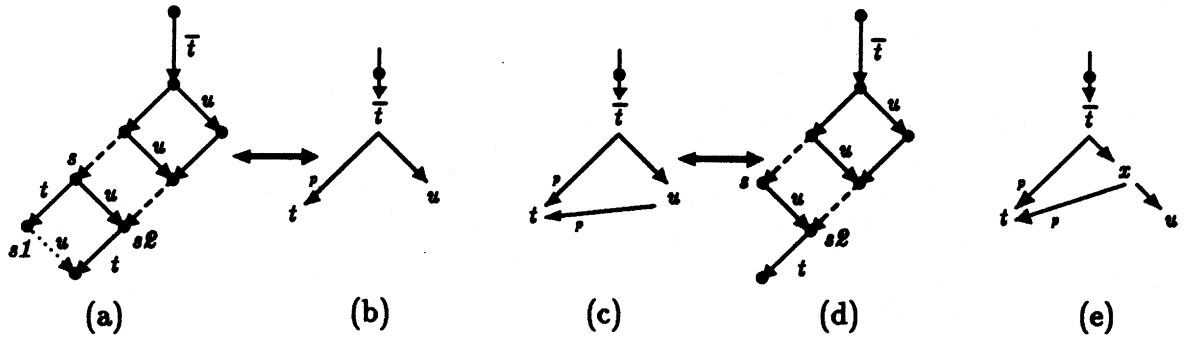


Figure 2.11: (a) A non-persistent state graph and (b) its corresponding STG. (c) By adding a persistency constraint, non-persistency can be eliminated from the state graph (d). (e) To maintain the concurrency between u and t, an internal transition x can be introduced.

We have developed the corresponding syntactic characterization on STG for non-persistency, as illustrated for a live STG in Fig. 2.11b: (Theorem 5.4)

> *In the state graph (Fig. 2.11a), transition u is non-persistent iff in the corresponding STG, t, u are concurrent and $\bar{t}Ru$.*

This can be appreciated by considering the corresponding hardware implementation: the course of action $\bar{t}Ru$ is implemented by a hardware element with $\bar{t}$ as one of its input transitions and u is a transition of its output signal. Concurrency between t and u implies that while the hardware element is reacting to $\bar{t}$ to cause u, t may be occurring simultaneously at the input of that hardware element. This is commonly known as a race condition in hardware circuits and can lead to malfunction.

A *persistency constraint* is an ordering constraint between two transitions, namely from u to t as illustrated in Fig. 2.11c, used for eliminating this non-persistent behavior. The resulting state graph in Fig. 2.11d is persistent.

A general condition of our design methodology is that transitions of *input* signals to the system are always assumed to be persistent. The reason for this assumption is that even if two transitions of input signals appear to be enabled in the same state, in the larger system comprising the original system and its environment, they may indeed be enabled in different states.

A STG is said to be persistent iff its state graph is persistent. Hence,

A STG is persistent iff for every non-input signal j, every transition of j caused by a transition x is ordered with $\bar{x}$.

A remark about Persistency.

Earlier, when introducing STGs as a form of interpreted Petri nets, we mentioned a reason why places are not interpreted as states of signals. This is indeed a fundamental limitation of our attempt to use LSFC nets to model behavior of logic circuits. As a matter of fact, STGs cannot model circuits which have hazards because these circuits correspond to non-persistent STGs. Consider the STG in Fig. 2.11b. Without a persistency constraint from u to t (as indicated in Fig. 2.11c), the following problem would arise. After transition $\bar{t}$ fires, each of its output places (not shown) gets a token. Since u and t are concurrent, the token on the left branch may move to the input place of t while u has not fired. Since t and $\bar{t}$ are transitions of the same signal (say j), what should really happen when t fires subsequently is that the token at the input place of u also disappears. This is because the presence of this token should be used to indicate the state of signal j after $\bar{t}$ fires. Since t has fired, changing the state of j again, this token must disappear.

It is clear that such a disappearing token is not allowed by the firing rule of Petri nets. Hence in order not to have to deal with this problem, we require that all transitions in a net (except those of input signals) satisfy the persistent constraint. The ramification of this restriction is that STGs cannot model arbiters and similar circuits with hazards. Note, however, that this does not include circuits with input choices which can still be specified by STGs (as illustrated by Fig. 2.2e). Hence, an arbitration can always be turned into an input choice by using an external arbiter.

2.3.4 A synthesis procedure

The following synthesis procedure summarizes the ideas discussed so far. From a textual description of a control circuit, we perform the following steps.

(1) Construct a STG specification according to the description.

(2) Check the syntax of the STG for liveness.

(3) Check the syntax of the STG for persistency. (If non-persistency exists, eliminate it by adding persistency constraints and/or additional internal signals.)

(4) Check the syntax of the STG for state assignment problems.

(5) Derive the equivalent FA of the STG and perform state assignment to produce a state graph. (If the STG is live then its state graph has a consistent state assignment.)

(6) Determine the network function from the state graph.

The logic circuit is simply the realization of the network function. The phase of transforming a live-persistent STG into a logic circuit can be done mechanically. However, transforming an initial STG specification into a live-persistent one may require interactions with the designer. Even though checking for liveness and persistency of STGs can be performed by simply checking the syntax of STGs, there is usually more than one way to eliminate deadlocks and non-persistency. For example, non-persistency in Fig 2.11b can be eliminated either (i) by adding a persistency constraint directly to the STG (Fig. 2.11d) or (ii) by first introducing an internal transition x and then adding the persistency constraint. In case (i), concurrency in the specification is reduced, thus reducing the number of transition sequences allowed. In the second case, concurrency is maintained at the cost of introducing more signal transitions (and hence more hardware). Whichever choice is better—limited concurrency vs. *external* behavioral equivalence—is to be decided by the designer.

We caution the reader that step (4) is another possible mechanism which may cause non-persistency in the state graph; it is a by-product of the state assignment process and has no relation to the sequencing specification of a STG. Nonetheless, we still can establish a syntactic characterization of this problem for STGs, as presented in Chapter 5. In Section 2.5, we present an example of this problem and discuss its implications.

2.4 Decomposition by Net Contraction

The above synthesis procedure simply consists of going from a live-persistent STG to a state graph then to a logic implementation. There is a simple method for decomposing the state graph into smaller subgraphs, from which an efficient implementation can be

obtained. This decomposition technique is based on a graph-theoretic operation called *contraction.* The general strategy is illustrated in Fig. 2.12 which shows the steps for the synthesis of a *trigger module*, a control module for implementing pipelined operation of a processign system.

The STG specification of this module (Fig. 2.12a) satisfies the syntactic properties of liveness and persistency. Its state graph is shown in Fig. 2.12b, from which an implementation can be obtained (Fig. 2.12e). This path a–b–e adheres to the synthesis steps described previously, where no decomposition is used. The alternate path a–c–d–e demonstrates our decomposition technique, as described below.

First, for each signal to be implemented in $\{I_a, O_r\}$, a *contracted net* is determined for it, as shown in Fig. 2.12c. In the STG, transitions of signal I_a are *caused* by transitions of signals I_r and O_r, but not O_a. Thus in the contracted net of I_a, transitions of O_a can be eliminated. Besides transitions of signal I_a, the contracted net of I_a only contains transitions of signals which *cause* transitions of I_a in the original STG. These signals $\{I_r, O_r\}$ form the *input set* of I_a, denoted as $I(I_a)$. In the implementation (Fig. 2.12e), signals in this set are input signals to the logic element I_a. A contracted net is obtained by removing unwanted transitions from the original STG, in such a way that the temporal relation between remaining transitions is preserved. The left STG in Fig. 2.12c is the contracted net of I_a, obtained by removing transitions of signal O_a; similarly, the right STG is the contracted net of O_r, obtained by removing transitions of signal I_r. The dashed arcs in the contracted nets of (c) indicate redundant causal constraints which can be removed.

It is important to note that in the contracted net of a signal j, only transitions of j are considered as *output*, other transitions in the input set of j, $I(j)$, are considered as *inputs* to the logic element j. The implication is that *since transitions in $I(j)$ are inputs, they are persistent.* Their persistency must be guaranteed by other part of the circuit which has its own corresponding contracted net.

Using exactly the same technique as before, each contracted net produces a state graph from which the logic equation for a signal can be determined. In this example, the final hardware circuits are the same whether decomposition is used or not. However, in general, the circuit obtained from decomposition is usually more efficient.

The dashed arrows from (a) to (c) and from (b) to (d) indicate net contractions and state graph contractions, respectively. The arrows from (b) to (d) indicate the fundamental reason this technique of decomposition yields a logic circuit with the same behavior as one obtained without decomposition: state graphs obtained from the contracted nets are themselves contracted versions of the state graph in (b). A *contracted state graph* is obtained by simply "ignoring" the uninteresting transitions.

The major net-theoretic results which serve as the basis of this decomposition technique are the following (Fig. 2.13a). Let Σ denote a LSFC net satisfying the one-token SM restriction, and Φ its equivalent FA. Let $\{\Sigma'_1, \ldots, \Sigma'_n\}$ be a set of contracted nets of Σ such that each Σ'_i contains a subset $T_i \subseteq T$, where T is the set of transitions of Σ. A contraction preserves the temporal relation if two transitions which are ordered (concurrent, in conflict) in the original STG remain so in the contracted net. For each Σ'_i, let Φ'_i be its equivalent FA. Then (Theorem 6.11)

> *If the contractions preserve the temporal relation, then every Φ'_i is a contracted state graph of Φ.*

The state graphs in $\{\Phi'_1, \ldots, \Phi'_n\}$ form a collection of concurrently operating state machines. The aggregate behavior of this collection is defined as the concurrent composition of all components, obtained by weaving them: $\Phi'_1 \| \ldots \| \Phi'_n$.

The following result establishes the behavioral equivalence between the LSFC net and its set of contracted nets. (Theorem 6.12)

> *If the causal relation of every pair of transitions in Σ (i.e. $\forall t, t' \in T : tRt'$) is present in some contracted net, then Φ and $\Phi'_1 \| \ldots \| \Phi'_n$ are identical.*

Since STGs are a syntactic subclass of LSFC nets, the above result can be applied directly to STGs, as indicated by Fig. 2.13b. The logic circuits obtained by two methods are equivalent in the sense that they have the same behavior, as expressed in terms of sets of transition sequences.

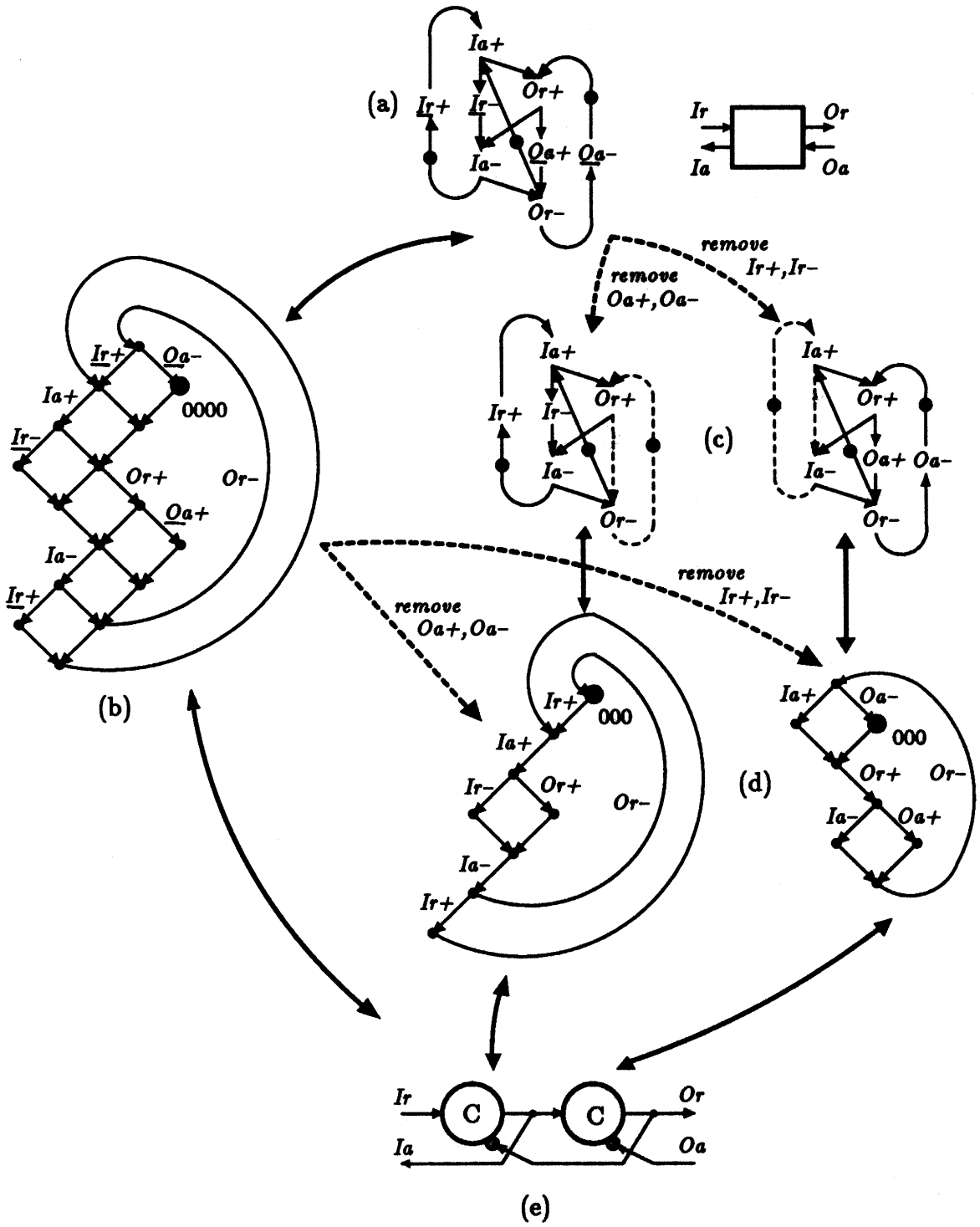


Figure 2.12: (a) STG specification of a trigger module. The final implementation can be obtained through the normal path a–b–e, or through decomposition by contraction a–c–d–e.

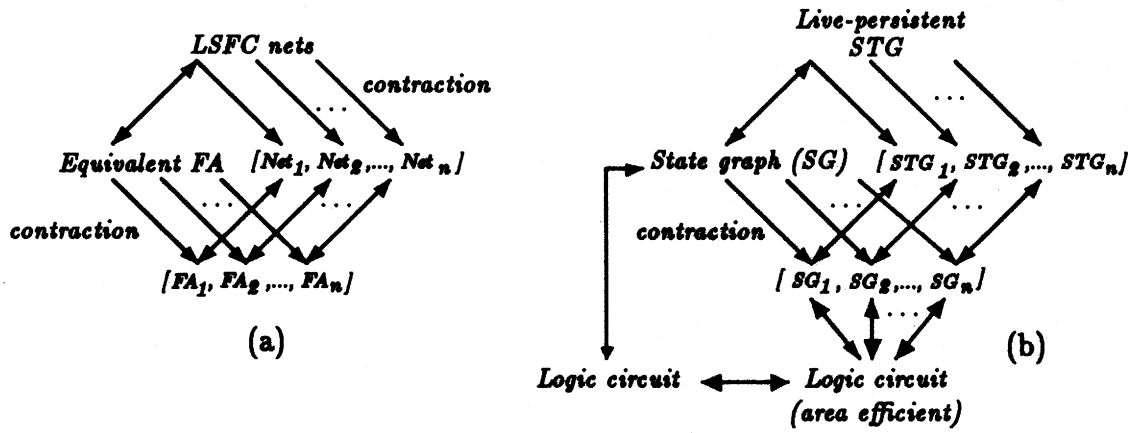


Figure 2.13: (a) The equivalent FA of contracted nets are identical to the contraction of the original FA. (b) Application of the result in (a) to STGs.

A remark about contraction vs. reduction

Earlier in this chapter, we mentioned Hack's reduction algorithm which, when applied to a LSFC net, produces a set of MG-components or SM-components. In addition, we have shown that the equivalent FA of a LSFC net can be obtained by weaving the equivalent FA of a set of covering SM-components.

The result stated above for net contraction indicates that the equivalent FA of a LSFC net can also be obtained by weaving the equivalent FA of a set of *contracted* nets which "cover" the original STG (in the sense defined above).

The main difference between these two methods of decomposition is that in reduction, a net is decomposed into subnets which are either state machines or marked graphs, while in contraction, subnets are not necessarily required to be state machines or marked graphs, even though it is possible. Subnets resulting from net contractions can be anything, depending on which transitions are eliminated. In fact, reduction can be considered as a special case of contraction: in a SM- (MG-) reduction, the set of transitions which remain all belong to the same SM- (MG-) component; by using contraction to eliminate other transitions, the desired SM- (MG-) component can be obtained. Note, however, that the contraction technique proposed only works for LSFC nets which satisfy the one-token SM restriction.

2.5 A problem with state assignment

Previously, we discussed conditions under which a STG has a *consistent* state assignment. Below, we will use an example to illustrate a side-effect of state assignment which can sometimes give rise to non-persistency in state graphs. In contrast, the kind of non-persistency which we described earlier is a result of the interaction between concurrent signal transitions, and therefore, it is directly related to the sequencing requirement specified by an STG. Thus, even though both phenomena may give rise to non-persistency in state graphs, their mechanisms are completely different. We will call this the *state assignment problem* and the other *non-persistency in STGs* to distinguish them.

Fig. 2.14a is a STG, being a simple cycle $t_1 Rt_2 \ldots Rt_n Rt_1$, with an initial marking M_0 in which place $\langle t_n, t_1 \rangle$ is marked. Its equivalent FA (Fig. 2.14b) is a simple cycle, represented by the sequence $s_0[t_1\rangle s_1[t_2\rangle \ldots s_{n-1}[t_n\rangle s_0$. Suppose that there exists some transition t_i, $1 < i < n$, such that the set $B = \{t_1, t_2, \ldots, t_i\}$ has the property that $\forall x \in T : x \in B \Leftrightarrow \bar{x} \in B$, i.e. B forms a subset which contains *both* the rising and falling transitions of a number of signals. Such a set is called a *complementary set.*

Then when state assignment is carried out, states s_0 and s_i will have the same binary representation (Fig. 2.14c). This is because if any transition t_j in B occurs, then $\bar{t}_j$ must also have occurred. Thus in the state graph, both t_1 and t_{i+1} are enabled in the same state and hence, are in direct conflict. There are two cases:

(a) If both t_1 and t_{i+1} are transitions of input signals, then they are persistent. (This is due to the previous assumption that external transitions be persistent.)

(b) If either t_1 or t_{i+1} or both are transitions of non-input signals then non-persistency results.

Case (b) above represents an undesirable situation and it cannot be fixed by a persistency constraint as discussed earlier. In order to eliminate this case of non-persistency, one has to introduce additional bit to the binary vectors to allow the distinction between states s_0 and s_i. This means that it may be necessary to add another *internal* signal to the set of signals J. For example, one can insert a new signal transition x anywhere in the chain $t_1 Rt_2 \ldots Rt_n$, thus ensuring that it no longer forms a complementary set. A specific example is given in Section 7.2.2 to illustrate this step.

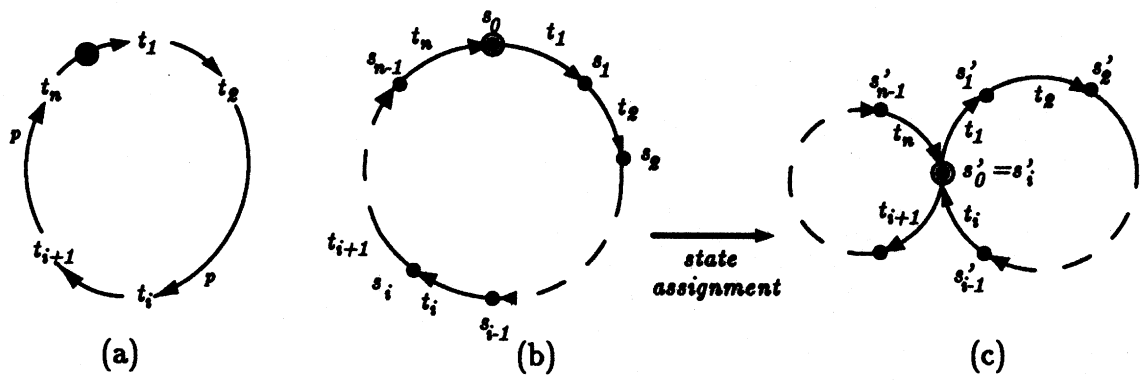


Figure 2.14: (a) A STG, (b) its equivalent FA and (c) the state graph resulting from state assignment.

The above problem also applies to *contracted nets* of a STG. Even if a STG contains no complementary sets, in a contracted net, new complementary sets may be created when transitions are removed from the STG. In which case, one can use the same technique of introducing additional internal signal to eliminate non-persistency in a contracted net. Chapter 7 provides an example for this.

2.6 Summary

In this chapter, we have outlined an approach for synthesis of self-timed control circuits from STG specifications. While transforming a live-persistent STG into a logic circuit can be done mechanically, ensuring liveness and persistency of STGs is a more complex task which may require interactions with the designer. Nonetheless, these interactions only mean that there are usually more than one way to guarantee liveness and persistency and choosing the optimal solution is the task left to the designer.

One added complication arises from our technique for decomposition which produces contracted nets from a STG. Of the three properties which one has to satisfy: liveness, persistency (of STG) and the state-assignment problem, we have proven in the thesis that (i) if liveness and persistency are met by a STG, then they are automatically met by its contracted nets, and (ii) even if a STG does not have any state-assignment problem, its contracted nets may due to the removal of transitions.

Chapter 3

Semantics and Temporal Relations of Nets

Petri net theory constitutes one active area of research in the description, modeling and analysis of concurrent systems. Petri nets allow the modeling of sequential and non-sequential behaviors of systems by providing two distinct types of elements for representing states and transitions. Even though Petri nets are a powerful model of concurrent systems, their analysis is often difficult due to the generality allowed in these specifications. We will restrict ourselves to a subclass of Petri nets called Free-Choice (FC) nets and will use them as specifications of systems which exhibit concurrent operations. Free-Choice nets represent an appropriate blend of concurrency and choice in specifying behaviors of systems, and at the same time can be analyzed without much difficulty.

In the previous chapter, we have argued for the use of firing sequence semantics of nets. This chapter provides the formal statements and proofs of the results presented earlier. Two new results described earlier for the class of LSFC nets are developed: (1) An algorithm for constructing a finite automaton directly from the *structure* of a net. Such a finite automaton corresponds precisely to the reachability graph of that net. (2) A characterization, based on the structure of a net, of a temporal relation on its set of transitions. The temporal relation can be partitioned into disjoint subsets of ordering, concurrency and conflict, each of which has a unique corresponding situation in the reachability graph.

The rest of this chapter is organized as follows. Section 2 gives a brief introduction to Petri net theory and summarizes important and recent results. The structure and firing

rules of Petri nets are defined; important subclasses of nets called marked graphs, state machines and free choice nets are identified. Then a number of results for LSFC nets are summarized, from works of Hack, Best, Thiagarajan and Voss. These are refered to as the *structure theory* of LSFC nets, one based on the static structure of the nets rather than on their dynamic behaviors as described by the firing rule.

Section 3 discusses a semantics of nets based on sets of firing sequences. We show that the set of firing sequences of a live safe free choice net can be determined directly from those of component state-machines by *weaving* them. This allows us to devise a simple algorithm for direct construction of the reachability graph from the structural components of a net.

Section 4 defines a binary relation called *temporal relation* in terms of net syntax and discusses a number of important properties. We then establish one-to-one correspondence between subsets of this relation (ordering, concurrency and conflict) with unique situations in the reachability graph.

3.1 Petri Nets

3.1.1 A Brief Introduction

This section provides a brief introduction to net theory. Most of the materials in this section are adapted from [24,41,44].

A Petri net is a triple $N = \langle P, T, F \rangle$ where

- $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$, and
- $F \subseteq (P \times T) \cup (T \times P)$ such that $dom(F) \cup range(F) = P \cup T$.

P is the set of *places*, T is the set of *transitions* and $P \cup T$ is the set of *elements* of N. The relation F is called the *flow relation*. In graphical representation, places are drawn as circles, transitions as bars and the flow relation as directed arcs. Let x be an element of N, then

- $\cdot x = \{y \in P \cup T \mid \langle y, x \rangle \in F\}$ is the *preset* of x,

- $x\cdot = \{y \in P \cup T \mid \langle x, y\rangle \in F\}$ is the *postset* of x.

For a place x, $\cdot x$ and $x\cdot$ are often referred to as sets of *input* and *output* transitions of x, respectively. Similarly, for a transition x, $\cdot x$ and $x\cdot$ are often referred to as sets of *input* and *output* places of x, respectively. This dot notation is extended to sets of elements in the obvious way. If $p \in P$ and $|p\cdot| > 1$ then p is called a ***shared*** place, where $|X|$ denotes the cardinality of set X. To facilitate the task of analysis and presentation, two common restrictions on a Petri net $N = \langle P, T, F\rangle$ are that it be *pure* and *simple*:[1]

- N is *pure* iff $\forall x, y \in P \cup T : \langle x, y\rangle \in F \Rightarrow \langle y, x\rangle \notin F$.
- N is *simple* iff $\forall x, y \in P \cup T : (\cdot x = \cdot y \wedge x\cdot = y\cdot) \Rightarrow x = y$. Another related property is *place-simple*: $\forall x, y \in P : (\cdot x = \cdot y \wedge x\cdot = y\cdot) \Rightarrow x = y$.

In light of the fact that we are mostly interested in behavior of nets in terms of firing sequences, place-simplicity is a more reasonable restriction, whereas simplicity is a rather stringent requirement in that it disallows the specification of certain types of choices. In summary, all nets considered are tacitly assumed to satisfy the following restrictions, unless explicitly stated otherwise:

Restriction 3.1 *All nets considered are finite, strongly-connected, pure and place-simple.*

We will be mainly concerned with three important subclasses of nets called *marked graphs* (MG), *state machines* (SM) and *free-choice* (FC) nets. A marked graph is a net in which each place has at most one input transition and at most one output transition: $\forall p \in P : |\cdot p|, |p\cdot| \leq 1$. Marked graphs represent the structure of deterministic concurrent systems. The dual notion of marked graphs is that of state machines. A state machine is a net in which each transition has at most one input place and at most one output place: $\forall t \in T : |\cdot t|, |t\cdot| \leq 1$. State machines represent the structure of nondeterministic sequential systems. A free-choice net is a net such that if any two transitions t_1 and t_2 share the same input place p, i.e. $|p\cdot| > 1$, then p is the unique input place of both t_1 and t_2: $\forall p \in p : |p\cdot| > 1 \Rightarrow \cdot(p\cdot) = \{p\}$.

[1]Note that this definition of "simple" is differently from Commoner's simple nets, which are called *asymmetric-choice* nets [44].

The states of a system whose structure is modeled by a net are represented by markings. A *marking* of a net N is a function $M : P \rightarrow \{0,1,2\ldots\}$. In diagrams, the marking M is indicated by placing $M(p)$ tokens (drawn as dots) on each place p. If a place contains tokens, it is *marked*, otherwise, it is *blank*. Function M is extended to sets of places such that $M(P) = \sum_{p \in P} M(p)$. A transition t is *enabled* at the marking M iff each input place of t is marked at M, i.e. $\forall p \in {\cdot t} : M(p) > 0$. When the enabled transition t *fires* at M, a new marking M' is reached which is given by

$$\forall p \in P : M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {\cdot t} - t{\cdot} \\ M(p) + 1 & \text{if } p \in t{\cdot} - {\cdot t} \\ M(p) & \text{otherwise} \end{cases}$$

The transformation of M into M' through the firing of t is denoted as $M[t\rangle M'$. Let T^* be the set of all finite-length sequences of symbols in T. Let M_0 be a marking of the net N and $\sigma = t_0 t_1 t_2 \ldots t_n \in T^*$ a sequence of transitions. Then σ is a *firing sequence* at M_0 iff there exist markings $M_1, M_2, \ldots, M_{n+1}$ of N such that $M_i[t_i\rangle M_{i+1}$ for $0 \leq i \leq n$. As usual, $M_0[\sigma\rangle M_{n+1}$ denotes the transformation of M_0 into M_{n+1} by firing σ at M_0. By convention, for every marking M of N, $M[\epsilon\rangle M$, where ϵ denotes the *empty* sequence.

The *forward marking class* of a marking M of a net N is denoted as $[M\rangle$ and is the smallest class of markings of N given by:

- $M \in [M\rangle$ and
- if $M' \in [M\rangle$ and for some $t \in T : M'[t\rangle M''$, then $M'' \in [M\rangle$.

A net $N = \langle P, T, F\rangle$ with an initial marking M_0 is usually represented as a quadruple $\Sigma = \langle P, T, F, M_0\rangle$. N is called the *underlying* net of Σ and usually denoted as N_Σ.

Two important behavioral properties of a net with an initial marking are liveness and safety.

Definition 3.2 *Let $\Sigma = \langle P, T, F, M_0\rangle$ be a net with an initial marking M_0. Then*

- *Σ is live iff $\forall M' \in [M_0\rangle, \forall t \in T : \exists M'' \in [M'\rangle$ such that t is enabled at M''.*
- *Σ is safe iff $\forall M' \in [M_0\rangle, \forall p \in P : M'(p) \leq 1$.*

3.1.2 Previous Results for Free-Choice nets

Given a Petri net with an initial marking, the firing rule can be applied to transform the net from one marking into a new marking by firing enabled transitions in the net. The *reachability graph* is a graph in which nodes are markings and arcs between nodes represent firings of transitions; this graph is a state transition graph which readily captures the behavior of the net. Although for general Petri nets, reachability graphs can be obtained by exhaustively firing enabled transitions at a marking to produce new markings, for the class of FC nets, much can be deduced directly from their structure without reference to any particular initial marking, provided that the net is live and safe. In this section, we summarize a number of relevant results for FC nets.

Reduction algorithms for Free-Choice nets.

Hack [24] devised two simple algorithms for structurally reducing a free choice net into its component marked graphs or state machines.

A marked graph *allocation* (MG allocation) over a free choice net $N = \langle P, T, F \rangle$ is a function $A : P \rightarrow T$ such that $\forall p \in P : A(p) \in p\cdot$. Thus for every place in a net, only one of its output transitions is allocated, the rest are called ***unallocated*** transitions. Let E_t and E_p denote the sets of eliminated transitions and places, respectively. The marked graph *reduction* (MG reduction) algorithm involves the following steps. $\forall p \in P, \forall t \in T$:

1. Delete all unallocated transitions: $p\cdot - \{A(p)\} \subseteq E_t$.
2. Delete places with all input transitions already deleted: $\cdot p \subseteq E_t \Leftrightarrow p \in E_p$.
3. Delete transitions with at least one input place deleted: $\cdot t \cap E_p \neq \emptyset \Leftrightarrow t \in E_t$.

Repeat steps 2 and 3 until they are no longer applicable. A marked graph resulting from the reduction is called a *MG-component.*

A state-machine allocation (SM allocation) over a free choice net $N = \langle P, T, F \rangle$ is a function $B : T \rightarrow P$ such that $\forall t \in T : B(t) \in \cdot t$. The state machine reduction (SM reduction) algorithm involves the following steps. $\forall p \in P, \forall t \in T$:

1. Delete all unallocated places: $\cdot t - \{B(t)\} \subseteq E_p$.

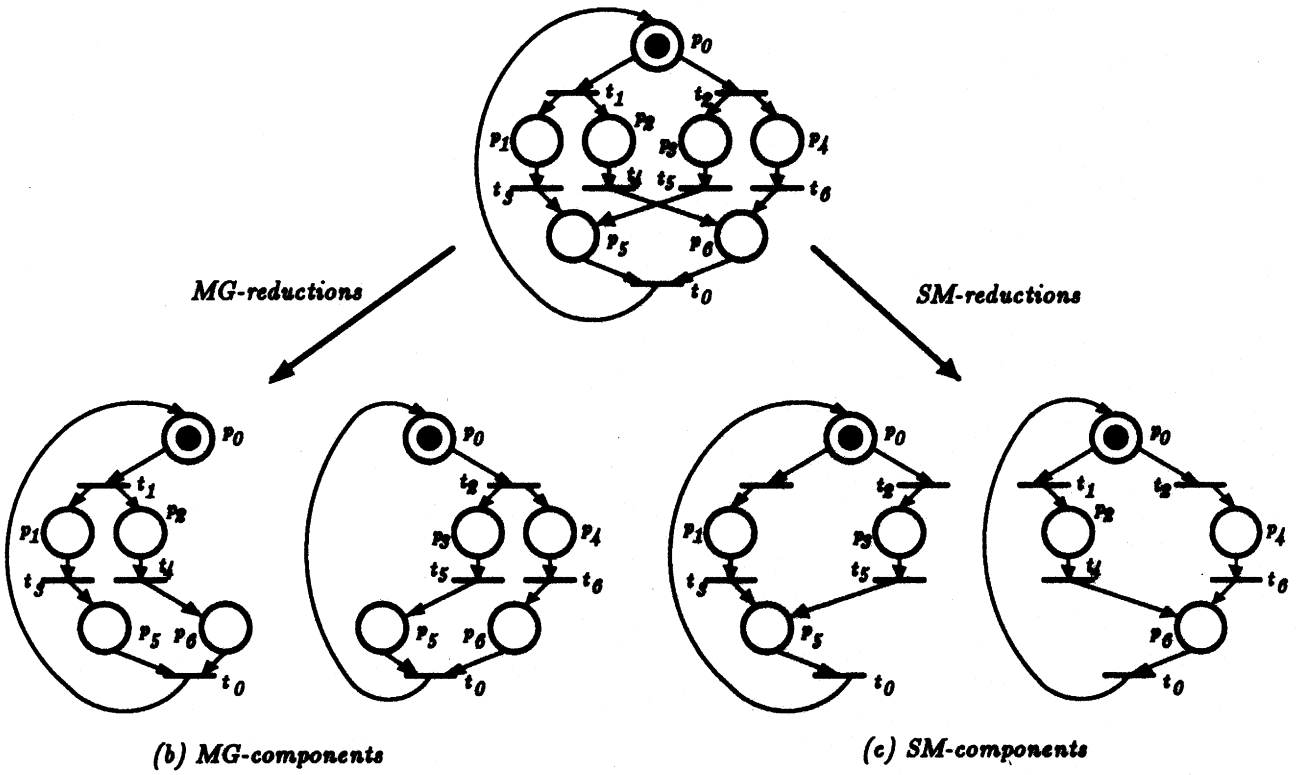


Figure 3.1: (a) A LSFC net (b) Its MG components resulting from MG-reductions and (c) Its SM components resulting from SM-reductions.

2. Delete transitions with all output places already deleted: $t\cdot \subseteq E_p \Leftrightarrow t \in E_t$.
3. Delete places with at least one output transition deleted: $p\cdot \cap E_t \neq \emptyset \Leftrightarrow p \in E_p$.

Repeat steps 2 and 3 until they are no longer applicable. A state machine resulting from the reduction is called a *SM-component*. An example (taken from [4]) of MG and SM reduction of a LSFC net is shown in Fig. 3.1.

Useful results for Free-Choice nets.

For FC nets, a number of useful results have been proven. They are stated below as preliminaries to further investigation. The following theorem is due to Hack, phrased in different form by [49].

Theorem 3.3 *Let $\Sigma = \langle P, T, F, M_0\rangle$ be a LSFC net and x an element of N_Σ. Then*

- *there exists a MG-component* $N_1 = \langle P_1, T_1, F_1 \rangle$ *of* N_Σ *such that* $x \in P_1 \cup T_1$,
- *there exists a SM-component* $N_2 = \langle P_2, T_2, F_2 \rangle$ *of* N_Σ *such that* $x \in P_2 \cup T_2$.

This theorem states that if a FC net is live and safe, then the set of MG-components resulting from MG-reductions *covers* the net. Alternatively, the set of SM-components resulting from SM-reductions also covers the net. On the other hand, if a FC net is either nonlive or unsafe, then some reduction does not cover the net or is empty, or some component is not strongly connected.

Another useful theorem from the theory of marked graphs [12] states that (a) for every marking M reachable from the initial marking M_0, there always exists a firing sequence σ which brings the net from M back to itself, and (b) any such firing sequence σ must fire all transitions in the net at least once and exactly the same number of times. This theorem is formalized below.

For a sequence $\sigma \in T^*$ and $T_1 \subseteq T$, $\sigma\lceil T_1$ denotes the projection (or restriction) of σ onto set T_1. If $t \in T$ is a transition then $\#(\sigma\lceil t)$ is the number of occurrences of t in σ. For a set of transitions T_1, $\#(\sigma\lceil T_1) = \sum_{t \in T_1} \#(\sigma\lceil t)$. For a net with an initial marking $\Sigma = \langle P, T, F, M_0 \rangle$, $FS(\Sigma)$ denotes the set of all firing sequences of Σ. The following theorem states that in a live-safe marked graph, (a) for every marking M reachable from M_0, there exists a firing sequence which brings the net from M back to itself, and (b) every such firing sequence contains at least one instance of each transition.

Theorem 3.4 *Let* $\Sigma = \langle P, T, F, M_0 \rangle$ *be a live-safe marked graph. Then,*

$$(a) \quad \forall M \in [M_0\rangle \; \exists \sigma \in T^* : M[\sigma\rangle M \quad \textit{and}$$
$$(b) \quad (\forall \sigma \mid M[\sigma\rangle M)(\forall t \in T) : \#(\sigma\lceil t) = k, \quad \textit{for some } k \geq 1.$$

There are a number of recent results concerning the structural properties of LSFC nets. We mention below two theorems from [49]; these are important for later developments in the thesis. Let M be a marking of a net N, and $P_1 \subseteq P$. Then $M\lceil P_1$ denotes the submarking obtained by restricting M to P_1.

Informally, the theorem below states that the behavior of a SM-component—as characterized by firing sequences—is not constrained in any way by the composite FC net. The set of firing sequences of a SM-component Σ_1 is the same as that of the composite net

restricted to the transitions in T_1. Thus a LSFC net can be considered as a set of state machines which operate concurrently and synchronize with each other occasionally.

Theorem 3.5 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net, and suppose $\Sigma_1 = \langle P_1, T_1, F_1, M_0^1 \rangle$ is a SM-component of Σ, where $M_0^1 = M_0 \lceil P_1$. Then*

$$FS(\Sigma_1) = FS(\Sigma) \lceil T_1,$$

where by definition $FS(\Sigma) \lceil T_1 = \{\sigma \lceil T_1 \mid \sigma \in FS(\Sigma)\}$.

The following theorem states that each MG-component in a LSFC net can be activated at some marking and it is a LS marked graph under that marking.

Theorem 3.6 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net and $N_1 = \langle P_1, T_1, F_1 \rangle$ be a MG-component of Σ. Then there exists a marking $M \in [M_0\rangle$ such that $\Sigma_1 = \langle P_1, T_1, F_1, M^1 \rangle$ is a live safe marked graph, where $M^1 = M \lceil P_1$.*

3.2 Firing Sequence Semantics of Free-Choice nets

One main result of this section is a theorem stating the relation between the behavior of a LSFC net and those of the component subnets constituting the original net. The behavior of nets will be characterized in terms of sets of firing sequences. We will show that these sets are *regular* by demonstrating a procedure for constructing their equivalent finite automata (FA). For nets, these FA correspond precisely to the reachability graphs whose vertices represent the markings of nets, and arcs between vertices the transitions from one marking to another due to the firing of some transition in the net. This theorem provides an algorithm for obtaining the equivalent FA directly from the *structure* of a LSFC net, instead of having to determine every marking from the firing rule.

3.2.1 Semantics

Theorem 3.5 states that the set of firing sequences of a SM-component Σ_1 is the same as that of the composite net restricted to the transitions in T_1. Thus a LSFC net can be

considered as a set of state machines which operate concurrently and synchronize with each other occasionally. Given this fundamental viewpoint, we can characterize the behavior of a LSFC net as a composition of firing sequences of the component state machines.

Before discussing the theorem, we present a convenient operator called *weave* for combining firing sequences of concurrently operating nets. For a net Σ, $FS(\Sigma)$ denotes its set of firing sequences.

Definition 3.7 *Let* $\Sigma_1 = \langle P_1, T_1, F_1, M_0^1 \rangle$ *and* $\Sigma_2 = \langle P_2, T_2, F_2, M_0^2 \rangle$ *be two nets. The* weave *of two sets of firing sequences* $FS(\Sigma_1)$ *and* $FS(\Sigma_2)$ *is given by*

$$FS(\Sigma_1) \parallel FS(\Sigma_2) = \{\sigma \in (T_1 \cup T_2)^* \mid \sigma\lceil T_1 \in FS(\Sigma_1) \ \wedge \ \sigma\lceil T_2 \in FS(\Sigma_2)\}.$$

Weaving is idempotent, commutative and associative. If $T_1 \cap T_2 = \emptyset$, weaving is exactly the *shuffle* of two sets of sequences. The weave operator has been used in [25,53] as a convenient means of describing "synchronized concurrency" between two sequences of events, such that distinct events in two sequences can occur concurently in any order, but their common events must occur in synchrony. In these formulations, the set of firing sequences is always accompanied by the set of events to form a pair $\langle T, FS \rangle$ called a *trace structure*, where $FS \subseteq T^*$. The inclusion of the set of transitions T is necessary in order for weave to be associative. In our formulation, the corresponding trace structure of a LSFC net $\Sigma = \langle P, T, F, M_0 \rangle$ is given by $\langle T, FS(\Sigma) \rangle$.

Theorem 3.3 shows that for a LSFC net, there exists a set of one-token SM-components which cover it. The existence of these components is essential to the correctness of the following theorem.

Theorem 3.8 *Let* $\Sigma = \langle P, T, F, M_0 \rangle$ *be a LSFC net,* $\{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$ *be a set of one-token SM-components which covers the net, where* $\Sigma_i = \langle P_i, T_i, F_i, M_0^i \rangle$ *and* $M_0^i = M_0\lceil P_i$, $1 \leq i \leq n$. *Then*

$$FS(\Sigma) = FS(\Sigma_1) \parallel FS(\Sigma_2) \parallel \ldots \parallel FS(\Sigma_n).$$

Proof. Let $FS' = FS(\Sigma_1) \| FS(\Sigma_2) \| \ldots \| FS(\Sigma_n)$. We need to show that $FS(\Sigma) = FS'$.

(a) $FS(\Sigma) \subseteq FS'$: Theorem 3.5 states that if Σ is a LSFC net then for every SM-component Σ_i, $1 \leq i \leq n : FS(\Sigma_i) = FS(\Sigma)\lceil T_i$. Hence, $\sigma \in FS(\Sigma) \Rightarrow \sigma\lceil T_i \in FS(\Sigma)\lceil T_i = FS(\Sigma_i)$, which further implies that $\sigma \in FS'$. Therefore $FS(\Sigma) \subseteq FS'$.

(*b*) $FS' \subseteq FS(\Sigma)$: We need to show that every sequence $\sigma \in T^*$ such that $\sigma\lceil T_i \in FS(\Sigma_i)$, $1 \leq i \leq n$, is a firing sequence of Σ, i.e. $\sigma \in FS(\Sigma)$. This is done by induction on $|\sigma|$.

Basis: $|\sigma| = 0$. Trivial.

Induction step: Assume that $\sigma = \bar{\sigma}t \in T^*$, where $\bar{\sigma}\lceil T_i \in FS(\Sigma_i)$, $1 \leq i \leq n$ and $\bar{\sigma} \in FS(\Sigma)$. Let M' be a marking of Σ such that $M_0[\bar{\sigma}\rangle M'$. We proceed to show that at M', if t is enabled in every SM-component which contains it, then t is also enabled in Σ. In which case, $\bar{\sigma}t$ is a firing sequence of Σ.

Let $p \in {\cdot}t$, then there must exist some SM-component Σ_j containing p, as the SM-components cover the net. Because Σ_j is a SM-component, it must also contain t. Since $\bar{\sigma}t\lceil T_j$ is a firing sequence of Σ_j, it follows that p must be marked at M'. By applying the same reasoning, we deduce that every input place of t must be marked. Hence at M', t is enabled in Σ. ∎

Even though no proof exists, we believe that the above theorem should apply also to a superclass of FS nets called *State-Machine Decomposable* (SMD) nets.

3.2.2 An algorithm for constructing reachability graphs

So far, we have demonstrated that the set of firing sequences of a LSFC net can be obtained by weaving those of its SM-components. It will be shown in this section that the set of firing sequences of any LSFC net is *regular* by demonstrating the existence of a finite automaton (FA) which accepts or generates such a set of firing sequences. The result of Theorem 3.8 above provides an algorithm for constructing a FA directly from the structure of the net.

We start first with a description of the algorithm itself. For a LSFC net Σ, $\Phi(\Sigma)$ denotes its equivalent FA (which is the same as its reachability graph).

Algorithm 3.9 *Let $\Sigma = \langle P, T, F, M_0\rangle$ be a LSFC net and $\{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$ be a set of one-token SM-components which covers the net. Then*

$$\Phi(\Sigma) = \Phi(\Sigma_1) \parallel \Phi(\Sigma_2) \parallel \ldots \parallel \Phi(\Sigma_n)$$

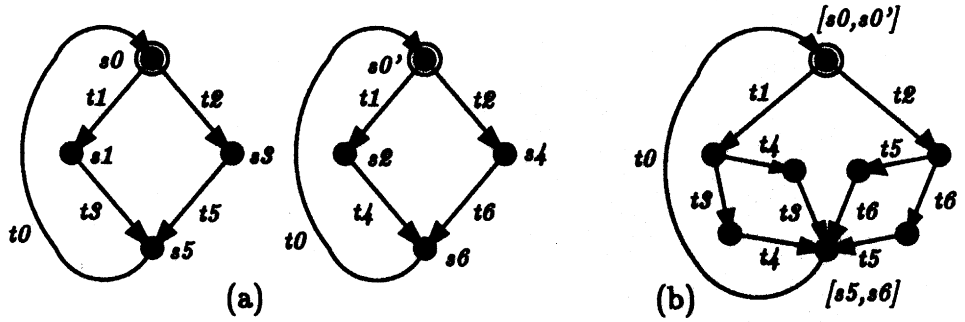


Figure 3.2: (a) the state machines of SM-components from Fig. 3.1, and (b) the weave of two state machines, yielding the reachability graph of the LSFC net in Fig. 3.1.

where $\Phi(\Sigma_i)\|\Phi(\Sigma_j)$ *denotes the weave of two state machines, as described subsequently.*

Fig. 3.2 shows construction of the equivalent FA of the LSFC net in Fig. 3.1 by weaving the state machines corresponding to the SM-components. The regular expressions for the component state machines are

$$E_1 = ((t_1t_3|t_2t_5)t_0)^* \text{ and } E_2 = ((t_1t_4|t_2t_6)t_0)^*.$$

The regular expression for the LSFC net is $E_1\|E_2$ and can be determined directly from the state graph in (b) as $E_1\|E_2 = ((t_1(t_3\|t_4)|t_2(t_5\|t_6))t_0)^*$.

Note that if Σ_i is a SM-component, then $\Phi(\Sigma_i)$ is the same as Σ_i itself, except for a few representational changes. In this case, $FS(\Sigma_i)$ can be represented by a regular expression involving the operators ***concatenation, union, and Kleene closure*** (denoted respectively by juxtaposition, bar |, and star $*$.) The construction algorithm in the above definition is more convenient and direct than the alternate approach of manipulating the set of firing sequencess and expressions, as suggested by Theorem 3.8. It is more convenient because all state machine components already exist; they need not be constructed from their sets of firing sequences.

First, we describe a more conventional representation of state machines. Recall that if a LSFC net $\Sigma = \langle P, T, F, M_0\rangle$ is a state machine then $\forall M \in [M_0\rangle : M(P) = 1$ and $\forall t \in T : |\cdot t| = |t \cdot| = 1$.

Definition 3.10 *Let* $\Sigma = \langle P, T, F, M_0\rangle$ *be a state machine. Then the finite automaton corresponding to* Σ *is given by* $\Phi(\Sigma) = \langle S, T, \delta, s_0, q\rangle$ *where*

- *For* $P = \{p_0, p_1, \ldots, p_n\}$, $S = \{\ \{p_0\}, \{p_1\}, \ldots, \{p_n\}\ \}$ *is the set of states,*
- $s_0 = \{p \mid M_0(p) = 1\}$ *is the initial state,*
- $q = \{s_0\}$ *is set of final states,*
- $\delta : S \times T \rightarrow \underline{S}$ *a partial function called transition function defined such that* $\forall t \in T$, $\forall p_1, p_2 \in P$:

$$\langle p_1, t\rangle \in F \ \wedge\ \langle t, p_2\rangle \in F \ \Leftrightarrow\ \delta(s_1, t) = s_2,$$

where $s_1 = \{p_1\}$, $s_2 = \{p_2\}$.

In graphical representation, states in S are drawn as small solid circles. The initial state is circled to distinguish it from the rest. The transition function is indicated by arcs between states, labeled with the appropriate transitions. Also by convention, $\forall s \in S$: $\delta(s, \epsilon) = s$. For clarity these arcs are omitted for every state s.

Let A be a regular set of firing sequences, then $FA(A)$ will be used to denote the corresponding finite automaton. For a LS net Σ, $FS(\Sigma)$ denotes its set of firing sequences and $FA(FS(\Sigma))$ denotes its finite automaton. Thus $FA(FS(\Sigma)) = \Phi(\Sigma)$. The weave of two finite automata is defined subsequently. In order to keep the definition general, the transition relation $\Delta \subseteq S \times T \times S$ will be used instead of the usual transition relation $\delta : S \times T \rightarrow S$.[2]

Definition 3.11 *Let* Σ_1 *and* Σ_2 *be nets with sets of firing sequences* $E_1 = FS(\Sigma_1)$ *and* $E_2 = FS(\Sigma_2)$ *and finite automata*

$$\Phi(\Sigma_1) \ =\ FA(E_1) \ =\ \langle S_1, T_1, \Delta_1, s_0^1, q_1\rangle$$
$$\Phi(\Sigma_2) \ =\ FA(E_2) \ =\ \langle S_2, T_2, \Delta_2, s_0^2, q_2\rangle$$

(*a*) *The weave of* $\Phi(\Sigma_1)$ *and* $\Phi(\Sigma_2)$ *is given by*

$$\Phi(\Sigma_1) \parallel \Phi(\Sigma_2) \stackrel{def}{=} FA(E_1 \| E_2).$$

(*b*) *The finite automaton corresponding to the weave of* E_1 *and* E_2 *is given by*

$$FA(E_1 \| E_2) = \langle S_1 \times S_2, T_1 \cup T_2, \Delta, \{s_0^1, s_0^2\}, q_1 \times q_2\rangle,$$

[2] A finite automaton with a transition *relation* is generally nondeterministic, while one with a transition *function* is deterministic. Converting from a nondeterministic finite automaton to a deterministic one is straightforward, using such well-known techniques as the *subset construction* method.

where Δ *is defined as follows:* $\forall a_1, a_2 \in S_1,\ \forall b_1, b_2 \in S_2,\ \forall t \in T_1 \cup T_2 \cup \{\epsilon\}$:

$$\langle a_1, t\lceil T_1, a_2\rangle \in \Delta_1 \wedge \langle b_1, t\lceil T_2, b_2\rangle \in \Delta_2 \Rightarrow \langle \{a_1, b_1\}, t, \{a_2, b_2\}\rangle \in \Delta.$$

The inclusion of the empty transition ϵ for all states s : $\langle s, \epsilon, s\rangle \in \Delta$ is essential for the concise characterization of a weave's state machine. By using the ϵ-transitions, one avoids the distinction between different cases corresponding to $t \in T_1 - T_2$, $t \in T_2 - T_1$ and $t \in T_1 \cap T_2$. The above definition is essentially due to [53].

3.3 Temporal Relations: Ordering, Concurrency and Conflict

Under the one-token SM restriction on a LSFC net, one can determine directly from its structure whether two transitions are ordered, concurrent, or in conflict. Since we are only interested in nets which are strongly connected directed graphs, these temporal characterizations will be different from those based on partial orders which correspond to acyclic graphs. We will first define these sub-relations based on the structure of LSFC nets and then discuss their meaning and correspondence to situations in reachability graphs.

As mentioned in Chapter 2, this syntactic characterization only works for LSFC nets with the additional restriction which we call *one-token SM restriction*, given again below. In fact, all results proven in this section assume this restriction.

Restriction 3.12 (One-token SM) *The class of LSFC nets considered is such that for a net in this class,* every *one of its SM-components contains exactly one token.*

3.3.1 Syntactic Characterization

Below we define symmetric binary relations **li**, **co**, **cf** and **dc** on the set of transitions T; they stand for *ordering, concurrent, conflict* and *direct-conflict*, respectively. A *simple path* in a net N is a path $x_1x_2\ldots x_n \subseteq N$ such that $x_i \neq x_j$, $1 \leq i \neq j \leq n$. A *simple cycle* in N is a simple path $x_1x_2\ldots x_n \subseteq N$ with $x_1 = x_n$. An element x belonging to a cycle or path Π is written as $x \in \Pi$.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
t_0	**id**						
t_1	**li**	**id**					
t_2	**li**	**dc**	**id**				
t_3	**li**	**li**	**cf**	**id**			
t_4	**li**	**li**	**cf**	**co**	**id**		
t_5	**li**	**cf**	**li**	**cf**	**cf**	**id**	
t_6	**li**	**cf**	**li**	**cf**	**cf**	**co**	**id**

Table 3.1: The temporal relation of the LSFC net in Fig. 3.1.

Definition 3.13 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net satisfying the one-token SM restriction. For distinct transitions $t, t' \in T$:*

(*a*) *t and t' are* ordered, *denoted as $\{t, t'\} \in$ **li**, iff there exists a simple cycle in Σ to which both t and t' belong.*

(*b*) *t and t' are* concurrent, *denoted as $\{t, t'\} \in$ **co**, iff $\{t, t'\} \notin$ **li** and there exists a MG-component of Σ to which both t and t' belong.*

(*c*) *t and t' are* in conflict, *denoted as $\{t, t'\} \in$ **cf**, iff*

1. *either $\{t, t'\} \notin$ **li** and there exists a SM-component to which both t and t' belong,*
2. *or there exists no SM-component or MG-component containing both t and t'.*

In the characterization of concurrency, the existence of a MG-component is required in order for two transitions to be concurrent (statement b). However, in case of conflicts, the existence of a SM-component is sufficient but not necessary for two transitions to be in conflict. Statement d2 indicates that two transitions are also in conflict if no reduction produces a subnet—be it a SM-component or a MG-component—which contains both of them. Intuitively, this makes sense, as two concurrent transitions must always occur together; this implies the existence of a such an MG-component. On the other hand, if two transitions are in conflict, the occurrence of one exclude that of the other. Hence it is possible that they do not both belong to any structural component of the net at all.

Applying the above definition to the FC net of Fig. 3.1, we can construct Table 3.1. Since the relations are symmetric, half of the table is implied by the other half. Note also that $\{t_4, t_5\}$ and $\{t_3, t_6\}$ belong to **cf**, but they do not both belong to any SM- or MG-component.

One special case of (c) is the *direct-conflict* relation: t and t' ($t \neq t'$) are in direct-conflict, denoted as $\{t, t'\} \in \mathbf{dc}$, iff $\cdot t \cap \cdot t' \neq \emptyset$. In the above definition, $\mathbf{dc} \subseteq \mathbf{cf}$: according to the FC hypothesis, there exists place p which is a unique shared place of t and t', therefore they cannot belong to the same simple cycle and furthermore, they must belong to the same SM-component. Also, define **id** as $\{\{x, x\}\} \subseteq T \times T$ and $\mathbf{id} \subseteq \mathbf{li}$. Another useful derived relation is $\mathbf{idc} = \mathbf{cf} - \mathbf{dc}$: If two transitions t and t' are in conflict but not in direct conflict, then $\{t, t'\} \in \mathbf{idc}$ and then they are said to be *in indirect conflict.*

In a LS marked graph $\Sigma = \langle P, T, F, M_0 \rangle$, a *cut* C is a maximal set of elements in $P \cup T$ which are pairwise not contained in the same simple cycle. That is, if we extend the binary relation $\mathbf{r} \in \{\mathbf{li}, \mathbf{co}, \mathbf{cf}\}$ to $(P \cup T) \times (P \cup T)$ then a cut $C \subseteq P \cup T$ is defined such that

$$\begin{cases} \forall x, y \in C : \{x, y\} \in \mathbf{co} \ \textit{and} \\ \forall z \in (P \cup T) - C, \exists x \in C : \{x, z\} \notin \mathbf{co}. \end{cases}$$

A *t-cut* $C_t \subseteq T$ consists of all transitions which can fire concurrently at some marking of the net. A *p-cut* $C_p \subseteq P$ consists of all places that can be marked at the same time. The following lemma states that in a LS marked graph, any LS marking M marks all places of a p-cut and leaves other places blank.

Lemma 3.14 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LS marked graph satisfying the one-token SM restriction. Then every life-safe marking $M \in [M_0\rangle$ marked all places of a p-cut C_p and none else. That is, there exists C_p such that*

$$M(p) = \begin{cases} 1 & \forall p \in C_p \\ 0 & \textit{otherwise} \end{cases}$$

Proof. Let $\{\Omega_1, \ldots, \Omega_n\}$ be the set of simple cycles in N_Σ where $\Omega_i = \langle P_i, T_i, F_i \rangle$. Since Σ is a marked graph, each place is uniquely identified by its unique input and unique output transitions. Hence a marked graph can be considered as a directed graph with vertices corresponding to transitions and arcs corresponding to places. Therefore by definition, a cut C_p is a maximal set of arcs such that no two belong to the same simple cycle. Hence,

every simple cycle in $\{\Omega_1, \ldots, \Omega_n\}$ must contain exactly one arc in C_p, i.e. $\forall i \in \{1, \ldots, n\}$: $|P_i \cap C_p| = 1$. Consider a marking M which marks all places in C_p and none else. Then under the one-token SM restriction, M is a live-safe marking because each simple cycle Ω_i contains exactly one token. ∎

It has been shown in [27] that for LS marked graphs with the property that every simple cycle contains exactly one token, there exists a unique equivalence class of life-safe markings: any two markings in this equivalence class are mutually reachable from one another. This equivalence class is precisely the set of all p-cuts $\{C_p\}$ of a LS marked graph, including those which are singleton sets ($|C_p| = 1$).

The above result for marked graphs can be generalized to LSFC nets in a straightforward manner. According to Theorem 3.6, every SM-component can be activated at some marking and it operates as a LS marked graph. Hence

Theorem 3.15 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net satisfying the one-token SM restriction. Then every life-safe marking $M \in [M_0\rangle$ marks all places of a p-cut C_p of some MG-component.*

Some Examples. The above definition of the temporal relation indicates that one can determine the relation between two transitions only by looking at the global structure of a LSFC net. In Fig. 4a, it appears that $C = \{p_1, p_2, p_3, p_4\}$ forms a p-cut in some net; however as shown in Fig. 4b, C is not a p-cut if there are two simple cycles —one containing t_1, t_2, the other t_3, t_4— which do not share some common transition. This is because p_2 and p_3 are contained in a simple cycle and are therefore ordered. C is a p-cut only when there exists at least one transition u shared by the cycles, as shown in Fig. 4c.

In another example (Fig. 4d), place p appears to be a *redundant place* [2] and could be removed without changing the behavior of the net. However, Fig. 4e shows that if p is removed then the simple cycle $t_1 p t_4 p_4 t_2 p_2 t_3 p_5 t_1$ would no longer exist; in which case t_1 and t_4 would become concurrent. Thus in this particular case, the removal of p does indeed change the behavior of the net drastically and therefore it is *not* redundant.

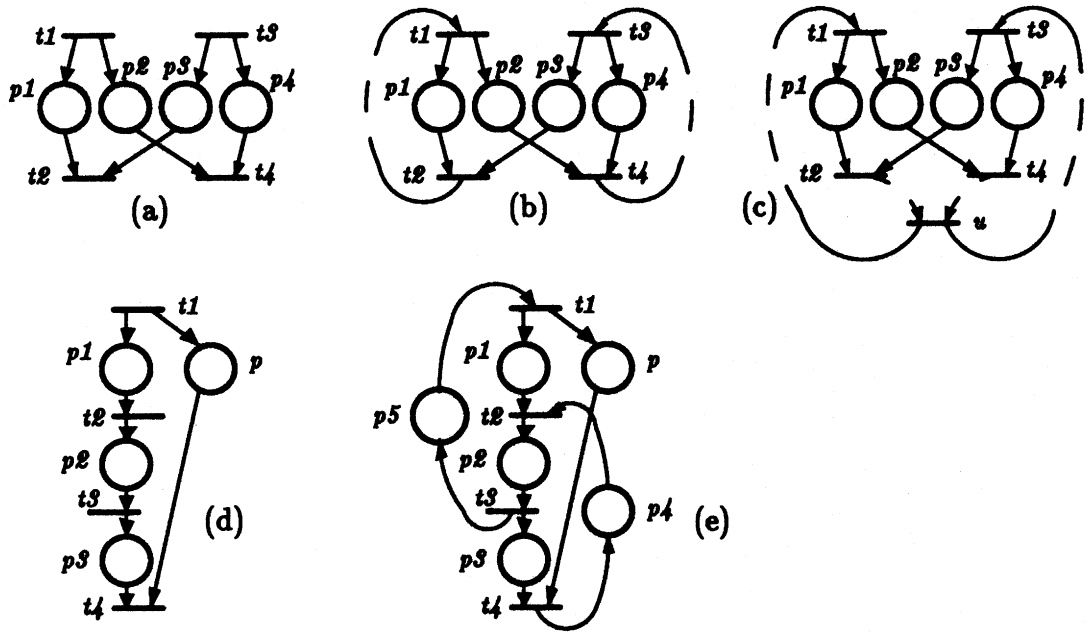


Figure 3.3: (a) $C = \{p_1, p_2, p_3, p_4\}$ appears to be a p-cut in some net. However (b) C is not a p-cut unless (c) there exists a shared transition u. (d) Place p appears to be a redundant place. However, in (e) it is not.

3.3.2 Partition of the Temporal Relation and Correspondence to Reachability Graphs

As shown next, the above syntactic characterizations partition the *temporal relation*, defined as $\mathbf{tr} = T \times T$, into disjoint subsets of ordering, concurrency and conflict, as is illustrated by Fig. 3.4. The subset **co** represents the *non-sequential* behavior, $\mathbf{li} \cup \mathbf{cf}$ represents the *sequential* behavior of the net. For LS marked graphs, $\mathbf{cf} = \emptyset$ and $\mathbf{tr} = \mathbf{li} \cup \mathbf{co}$. For LS state machines, $\mathbf{co} = \emptyset$ and $\mathbf{tr} = \mathbf{li} \cup \mathbf{cf}$.

Theorem 3.16 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net satisfying the one-token SM restriction, and $\mathbf{tr} = T \times T$ be called the* temporal relation. *Then* **li**, **co** *and* **cf** *partition* **tr** *into disjoint subsets:*

(*a*) $\mathbf{li} \cup \mathbf{co} \cup \mathbf{cf} = \mathbf{tr}$, *and*
(*b*) $\mathbf{li} \cap \mathbf{co} = \mathbf{li} \cap \mathbf{cf} = \mathbf{co} \cap \mathbf{cf} = \emptyset$.

Proof. Statement (a) trivially follows from the definitions, and so do $\mathbf{li} \cap \mathbf{co} = \emptyset$ and $\mathbf{li} \cap \mathbf{cf} = \emptyset$. It remains to show that $\mathbf{co} \cap \mathbf{cf} = \emptyset$. To establish this fact, it suffices to

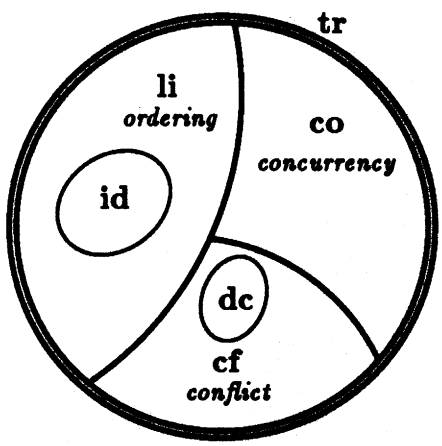


Figure 3.4: Partitions of the temporal relation **tr**.

show that in a LSFC net satisfying the one-token SM restriction, if two transitions do not belong to a simple cycle then they cannot belong to some MG-component ***and*** some SM-component at the same time. (In case two transitions are in conflict by not belonging to any structural component at all, it is trivial to see that they cannot be concurrent.)

Let N_Σ denote the underlying structure of Σ. First, note that if two transitions t and t' *do* belong to a simple cycle in N_Σ then t and t' must belong to a simple cycle in some MG-component $N_1 = \langle P_1, T_1, F_1 \rangle$ ***and*** in some SM-component $N_2 = \langle P_2, T_2, F_2 \rangle$, because both MG and SM reduction cover the net. Thus, if t and t' do not belong to any simple cycle in N, they do not belong to a simple cycle in any MG-component or SM-component. In this case, we show that if t and t' belong to both N_1 and N_2 then the net is either unsafe or nonlive.

If $t, t' \in T_1$ but no simple cycle containing both of them, then there must exist some $u \in T_1$ such that $t, u \in \Omega_1$ and $t', u \in \Omega_1'$, where Ω_1, Ω_1' are distinct simple cycles in N_1 (Fig. 3.5). If $t, t' \in T_2$ but no simple cycle containing both of them, then there must exist some $p \in P_2$ such that $t, p \in \Omega_2$ and $t', p \in \Omega_2'$, where Ω_2, Ω_2' are distinct simple cycles in N_2 (Fig. 3.5).

Since N_1 is a MG-component of a LSFC net, according to Theorem 3.6, there exists a marking which activates N_1 and N_1 will operate as a live-safe marked graph. Since t and t' do not belong to the same simple cycle, according to Theorem 3.15, in N_1 there exists a p-cut which corresponds to a marking $M \in [M_0\rangle$ under which both t and t' are enabled. Hence at M, all input places of t and t' are marked. Since the cycles Ω_2 and Ω_2' in N_2

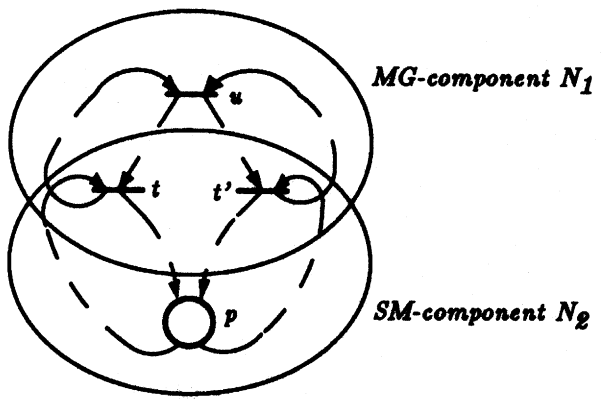


Figure 3.5: Proof of Theorem 3.16.

contain transitions t and t' respectively, they must each contains one input place of t or t'. Since all input places of t and t' are marked, the SM-component N_2 contains at least two tokens. Under the one-token SM restriction, this would lead to unsafeness. On the other hand, if N_2 contains only one token then only one of t, t' can possibly be enabled; this implies that u cannot be enabled subsequently, leading to a deadlock. ∎

The above classifications of the temporal relation are based on the syntax of LSFC nets. We now attempt to connect these classifications to the normal understanding of ordering, concurrency and conflict in terms of firing sequences. It will be shown that for LSFC nets, there exists a unique correspondence between an element of the temporal relation and a situation in a reachability graph of a net.

The following theorem gives the characterization of ordering, concurrency and conflict in terms of reachability graphs, or equivalently, firing sequences. Fig. 3.6 contains graphs illustrating these cases, in which vertices representing markings, solid arcs transitions between markings. The dotted arcs indicate explicitly those transitions which cannot occur; dashed arcs indicate a path between two markings.

A *cycle* in a reachability graph is characterized by $M_1[t_1\rangle M_2[t_2\rangle \ldots [t_{n-1}\rangle M_n$ where $M_1 = M_n$; $\sigma = t_1 t_2 \ldots t_{n-1}$ is a firing sequence such that $M_1[\sigma\rangle M_1$. If furthermore $M_i \neq M_j$, $1 \leq i \neq j \leq n$ then it is a *simple cycle*; in this case, σ is called the firing sequence corresponding to a simple cycle. In the following theorem and its proof, we use σ to denote such a firing sequence.

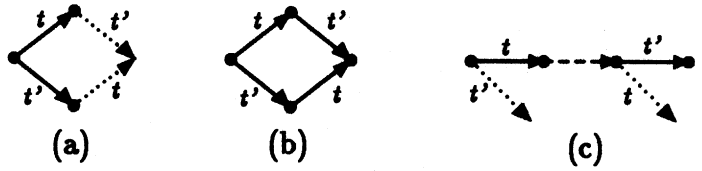


Figure 3.6: t, t' are (a) in direct-conflict (**dc**) (b) concurrent (**co**) (c) ordered or in indirect-conflict (**li** $\cup$ **idc**).

Theorem 3.17 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net satisfying the one-token SM restriction. Then $\forall t, t' \in T$:*

$$
\begin{array}{lll}
(a)\ \{t,t'\} \in \mathbf{dc} & \Leftrightarrow & \exists M \in [M_0\rangle : M[t\rangle \wedge M[t'\rangle \ \wedge \ \neg M[tt'\rangle \wedge \neg M[t't\rangle \\
(b)\ \{t,t'\} \in \mathbf{co} & \Leftrightarrow & \exists M \in [M_0\rangle : M[tt'\rangle \wedge M[t't\rangle \\
(c)\ \{t,t'\} \in \mathbf{li} \cup \mathbf{idc} & \Leftrightarrow & \nexists M \in [M_0\rangle : M[t\rangle \wedge M[t'\rangle \\
(d)\ \{t,t'\} \in \mathbf{idc} & \Leftarrow & (\nexists M \in [M_0\rangle : M[t\rangle \wedge M[t'\rangle) \wedge (\nexists \sigma \in T^* : t,t' \in \sigma),
\end{array}
$$

where σ is a firing sequence corresponding to a simple cycle in the reachability graph of Σ.

Parts (a) and (b) state that if two transitions are either concurrent or in direct conflict, then there is a marking in which both are enabled. Part (c) states that if two transitions are either ordered or in *indirect conflict* then they are never enabled in the same marking. Thus by inspecting the local structure of the reachability graph, it is possible to determine when two transitions are either concurrent or in direct conflict. However, ordering cannot be distinguished from indirect conflict. Part (d) further states that in order to discern these two cases, the global structure of the reachability graph needs be inspected: for every two transitions t and t', if there exists no marking M in which both transitions are enabled and furthermore, there exists no firing sequence σ corresponding to a simple cycle in the reachability graph such that $t, t' \in \sigma$, then t and t' are in indirect conflict.

Proof of Theorem 3.17. Part (a). ($\Rightarrow$) If $\{t,t'\} \in \mathbf{dc}$ then in N_Σ, there exists no simple cycle containing both t and t', and there exists some SM-component N_2 such that $t, t' \in T_2$. Every such SM-component must contain exactly one token. Furthermore according to the FC hypothesis, $\exists p \in P_2 : p\cdot = \{t,t'\}$ and $\exists M \in [M_0\rangle : M(p) = 1$. Hence $M[t\rangle$ and $M[t'\rangle$. Moreover, since the firing of either t or t' will remove the token in p, $M[tt'\rangle$ and $M[tt'\rangle$ cannot be true.

($\Leftarrow$) If $M[t\rangle \wedge M[t'\rangle$, and tt' and tt' are not firing sequences then it must be the case that $\cdot t \cap \cdot t' \neq \emptyset$. For suppose that $\cdot t \cap \cdot t' = \emptyset$. Then since $M[t\rangle \wedge M[t'\rangle$, all places in $\cdot t \cup \cdot t'$

must contain tokens. However this implies that $M[tt'\rangle \wedge M[t't\rangle$, a contradiction. Therefore, it must be the case that $\exists p \in P : {\cdot t} \cap {\cdot t'} = \{p\}$ and hence $\{t, t'\} \in \mathbf{dc}$.

Part (b). Note that $M[tt'\rangle \wedge M[t't\rangle \Rightarrow M[t\rangle \wedge M[t'\rangle$ according to the definition of firing sequences.

($\Rightarrow$) If $\{t, t'\} \in \mathbf{co}$ then in N_Σ there exists no simple cycle containing both t and t', and there exists some MG-component N_1 such that $t, t' \in T_1$. Since t and t' belong to different simple cycles of a marked graph N_1, ${\cdot t} \cap {\cdot t'} = t \cdot \cap t' \cdot = \emptyset$ and furthermore, ${\cdot t} \cap {\cdot t'} \subseteq C_p$, where C_p is a p-cut in N_1. Therefore, according to Lemma 3.14, $\exists M \in [M_0\rangle : M(p) = 1\ \forall p \in {\cdot t} \cup {\cdot t'}$. Clearly, at M both t and t' are enabled independently, thus proving the necessary condition.

($\Leftarrow$) First, we show that ${\cdot t} \cap {\cdot t'} = t \cdot \cap t' \cdot = \emptyset$. Suppose that ${\cdot t} \cap {\cdot t'} \neq \emptyset$, then due to the FC hypothesis, $\exists p \in P : {\cdot t} \cap {\cdot t'} = \{p\}$. Then according to Part (a), tt' and $t't$ cannot be firing sequence—a contradiction. Hence ${\cdot t} \cap {\cdot t'} = \emptyset$. From ${\cdot t} \cap {\cdot t'} = \emptyset$, using the fact that the net is safe, we show that $t \cdot \cap t' \cdot = \emptyset$. For if $t \cdot \cap t' \cdot \neq \emptyset$ then $\exists p \in P : p \in t \cdot \cap t' \cdot$. Since both t and t' are enabled, they can fire concurrently, placing two tokens in p and the net is unsafe.

Now we proceed to show that t and t' must belong to distinct simple cycles of some MG-component N_1. t and t' cannot both belong to any simple cycle Ω, for if they do then Ω must contain at least two tokens, each from an input place of t and t'; under the one-token SM restriction, this leads to unsafe behavior. Hence according to Theorem 3.16, they must belong to distinct simpe cycles of either a MG-component N_1 or a SM-component N_2. However, if N_2 existed, it would contain at least two tokens, one from each input place of t and t'. This causes unsafe behavior. Therefore they must belong to distinct cycles of N_1 and hence by definition, $\{t, t'\} \in \mathbf{co}$.

Part (c). From parts (a) and (b), it follows immediately that

$$\{t, t'\} \in \mathbf{co} \cup \mathbf{dc} \Leftrightarrow \exists M \in [M_0\rangle : M[t\rangle \wedge M[t'\rangle,$$

which means that if two transitions are either concurrent or in direct conflict, then there is a marking in which both are enabled. To prove (c), note that $(\nexists M \in [M_0\rangle : M[t\rangle \wedge M[t'\rangle) \Leftrightarrow \{t, t'\} \notin \mathbf{co} \cup \mathbf{dc} \Leftrightarrow \{t, t'\} \in \mathbf{tr} - \mathbf{co} \cup \mathbf{dc} = \mathbf{li} \cup (\mathbf{cf} - \mathbf{dc}) = \mathbf{li} \cup \mathbf{idc}$.

Part (d). From (c), we have $\not\exists M : M[t\rangle \wedge M[t'\rangle \Rightarrow \{t,t'\} \in \mathbf{li} \cup \mathbf{idc}$. We claim that $(\not\exists \sigma \in T^* : t,t' \in \sigma) \Rightarrow \{t,t'\} \in \mathbf{cf}$. The result follows immediately.

To verify the claim, note that if $\{t,t'\} \in \mathbf{li} \cup \mathbf{co}$ then there exists a MG-component N_1 containing both t and t'. According to Theorem 3.4, $\forall \sigma \in T_1^* \mid (\exists M \in [M_0\rangle : M[\sigma\rangle M) :$ σ must contain all transitions in T_1. Since $t,t' \in T_1$, it follows that $t,t' \in \sigma$. Also, since $T_1 \subseteq T$, we have $\{t,t'\} \in \mathbf{li} \cup \mathbf{co} \Rightarrow \exists \sigma \in T^* : t,t' \in \sigma$. Or equivalently,

$$(\not\exists \sigma \in T^* : t,t' \in \sigma) \Rightarrow \{t,t'\} \notin \mathbf{li} \cup \mathbf{co} \Leftrightarrow \{t,t'\} \in \mathbf{cf}.$$

∎

Chapter 4

Signal Transition Graphs

In this chapter, we introduce the Signal Transition Graph model. STGs correspond to the class of LSFC nets with interpreted transitions: transitions in nets are interpreted as *transitions of signals* in logic circuits. Most results for LSFC nets developed in the last chapter apply directly to STGs. Note that we only deal with LSFC nets which satisfy the one-token SM restriction.

We start out in Section 1 by giving the syntax and semantics of STGs. Briefly, a STG is a formal behavioral specification of a control circuit from which a set of transition sequences can be generated. Such a set has an equivalent finite automata representation called a state graph. From state graphs, one can determine the *network function*, which is a collection of logic functions describing the behavior of signals in the circuit. Section 2 discusses *state assignment*, the procedure for obtaining state graphs from STGs. Section 3 touches upon the area of composition of control modules.

4.1 Syntax and Semantics

Signal Transition Graphs are a formal behavioral specification of control circuits whose operation may involve ordering, concurrency and conflicts. A digital circuit (or network) is an interconnection of *logic elements*, each having one output terminal and a number of input terminals. Every input terminal is connected to either an input terminal of the entire network, or to an output terminal of another logic element in the network. The set of all

terminals of a network is called the set of *signals*. Let J denote such a set of signals, then J can be partitioned into the set of *input* signals J_I, the set of *output* signals J_O and the set of *internal* signals J_N. It is sometimes convenient to use the set of *non-input signals*, $J_{NI} = J_N \cup J_O \neq \emptyset$ to reflect the intention that changes in input signals are exogenous while changes in non-input signals are caused by the network.

The set of *signal transitions* is defined as $T = J \times \{+, -\}$. For every signal $j \in J$, there is a pair of signal transitions $\{j_+, j_-\}$ associated with it. We also adopt the notation that if t is used to denote j_+ then $\bar{t}$ denotes j_- and vice versa. When the direction of transition is not important, j_* is used to denote a transition of signal j (either j_+ or j_-). The set of signal transitions T is likewise partitioned into $T_I = J_I \times \{+, -\}$, $T_N = J_N \times \{+, -\}$ and $T_O = J_O \times \{+, -\}$. In order to distinguish between input and non-input signal transitions in a graphical representation, transitions in T_I are underlined.

4.1.1 Signal Transition Graphs

A Signal Transition Graph (STG) is a Petri net in which transitions are identified with signal transitions in a network whose set of signals is J.

Definition 4.1 (Signal Transition Graphs) *Let J be a set of signals of a network. A* Signal Transition Graph *defined on J is a Petri net $\Sigma_J = \langle P, T, F, M_0 \rangle$ with $T = J \times \{+, -\}$.*

Furthermore, a Signal Transition Graph is said to be well-formed *if (i) it is a LSFC net satisfying the one-token SM restriction and (ii) if a place p has more than one output transition, then all output transitions of p must be transitions of* input *signals, i.e.* $\forall p \in P : |p\cdot| > 1 \Rightarrow p\cdot \subseteq T_I$.

In the sequel, well-formed STGs will be referred to simply as STGs unless stated otherwise. Obviously, in a STG, $|T| = 2k$ where k is a positive interger. The restriction $\forall p \in P : |p\cdot| > 1 \Rightarrow p\cdot \subseteq T_I$ indicates that all signal transitions which are in direct conflict must be input ones, i.e. free choices are *input choices*. This allows STGs to specify the behavior of circuits whose operation involves sequentiality, deterministic concurrency and nondeterministic input choices.[1]

[1] In Chapter 8, we will extend the syntax of STG to allow the specification of *internal choices*. In these cases, one element from a set of output transitions of a place is chosen to be fired depending on the holding

As mentioned in Chapter 2, the graphical representation of STGs will differ from Petri nets in two accounts: (i) A transition in STGs is not depicted as a bar but by its name instead. (ii) Any place p in STGs with one input and one output transition, e.g. $\cdot p = \{t_1\}$ and $p\cdot = \{t_2\}$, will not be drawn. An arc going directly from t_1 to t_2 will be drawn instead: $t_1 \rightarrow t_2$. Such an arc is an instance of the *causal relation* $R \subseteq T \times T$, such that $t_1 R t_2 \Leftrightarrow \exists p \in P : \langle t_1, p\rangle \in F \wedge \langle p, t_2\rangle \in F$.

4.1.2 State Graphs

The semantics of STGs are given in terms of sets of transition sequences. STGs define sets of transition sequences which have equivalent FA representations called state graphs. These can be obtained by applying the results for LSFC nets described in Chapter 3.

For a FA Φ generated from a LSFC net Σ, there is a corresponding *state graph*, denoted by Φ_J generated from the STG Σ_J, where Σ_J is an interpretation of the LSFC net Σ. State graphs are defined as follows.

Definition 4.2 (State Graphs) *Let $J = \{j_1, j_2, \ldots, j_n\}$ be a set of signals of a circuit. A state graph defined on J is given by $\Phi_J = \langle S, T, \delta, s_0\rangle$,[2] where*

- *S is the set of states, defined as $S = \{s \mid s : J \rightarrow \{0,1\}\}$; every $s \in S$ is a function $s : J \rightarrow \{0,1\}$ and $\langle s(j_1), s(j_2), \ldots, s(j_n)\rangle$ is a binary vector of signal values in state s.*
- *$s_0 \in S$ is the initial state of the circuit.*
- *$T = J \times \{+,-\}$ is the set of signal transitions.*
- *$\delta : S \times T \rightarrow S$ is a partial function called the* transition function, *having the property that $\forall s, s' \in S, \forall t \in T$ such that $\delta(s,t) = s'$:*

$$\begin{array}{l} \textit{if } t = j_+ \textit{ then } s(j) = 0 \textit{ and } s'(j) = 1 \\ \textit{if } t = j_- \textit{ then } s(j) = 1 \textit{ and } s'(j) = 0. \end{array}$$

States s and s' are called adjacent; s' is a next-state of s.

---

of a certain condition in the circuit; the choices are no longer nondeterministic.

[2]Here, instead of describing a state graph as a 5-tuple $\langle S, T, \delta, s_0, q\rangle$, we adopt the convention to drop q (the set of final states) if $q = \{s_0\}$.

As noted in Chapter 2, in the construction of an equivalent FA from a set of sequences, a state is an abstract concept, defined as an equivalence class of sequences with the same postfixes; there is no apparent relationship between a state and the transitions enabled in it. However, for state graphs obtained from a STG, not only do we require that it be a FA equivalent to the set of sequences defined by a STG, but we also require that *states be interpreted as binary vectors representing the values of signals in a circuit.* There is a direct connection between states and transitions: states are vectors of values of a set of signals, whereas transitions are transitions of the same set of signals.

In the above definition, the transition function is extended to $\delta : S \times T^* \rightarrow S$ such that for $t \in T$ and $\sigma \in T^*$, $\delta(s, t\sigma) = \delta(s', \sigma)$ where $s' = \delta(s, t)$. Also, $\forall s \in S$, $\delta(s, \epsilon) = s$. By considering states as equivalent to markings, we can adopt the notations in Chapter 2: $s' = \delta(s,t) \Leftrightarrow s[t\rangle s'$; $s' = \delta(s,\sigma) \Leftrightarrow s[\sigma\rangle s'$; $[s\rangle$ denotes the set of states reachable from s, *etc.* Graphically, $s[t\rangle s'$ is represented as $\overset{s}{\bullet}\xrightarrow{t}\overset{s'}{\bullet}$; $s[\sigma\rangle s'$ as $\overset{s}{\bullet}\xrightarrow{\sigma}\overset{s'}{\bullet}$. A transition t is said to be *enabled* in state s if $s[t\rangle$; an enabled transition may occur or *fire.*

We denote the state graph of a STG Σ_J as Φ_J. The notions of ordering, concurrency and conflict in STGs and their correspondence in state graphs are exactly the same as those for LSFC nets. The following result is the restatement of Theorem 2.2 with *markings* replaced by *states.*

Theorem 4.3 *Let* $\Sigma_J = \langle P, T, F, M_0\rangle$ *be a STG and* $\Phi_J = \langle S, T, \delta, S_0\rangle$ *its state graph.* $\forall t, t' \in T$*:*

$$
\begin{array}{lll}
(a)\ \{t,t'\} \in \mathbf{dc} & \Leftrightarrow & \exists s \in [s_0\rangle : s[t\rangle \wedge s[t'\rangle \ \wedge\ \neg s[tt'\rangle \wedge \neg s[t't\rangle \\
(b)\ \{t,t'\} \in \mathbf{co} & \Leftrightarrow & \exists s \in [s_0\rangle : s[tt'\rangle \wedge s[t't\rangle \\
(c)\ \{t,t'\} \in \mathbf{li} \cup \mathbf{idc} & \Leftrightarrow & \nexists s \in [s_0\rangle : s[t\rangle \wedge s[t'\rangle \\
(d)\ \{t,t'\} \in \mathbf{idc} & \Leftarrow & (\nexists s \in [s_0\rangle : s[t\rangle \wedge s[t'\rangle) \wedge (\nexists \sigma \in T^* : t, t' \in \sigma)
\end{array}
$$

where σ *is a firing sequence corresponding to a simple cycle in* $\Phi(\Sigma_J)$.

We will return to more discussion of properties of state graphs in the next chapter. In order to give a complete picture of our direction, we describe next the *network function*, being a set of Boolean equations derived from the state graph. The network function describes the states of every signal in a digital network.

4.1.3 Network functions: Implementations of state graphs.

Similar to Muller [36], we define a digital network as an interconnection of a finite number of *logic elements* of unbounded delays and containing no clocks, i.e. operation is asynchronous. However, we extend his autonomous networks (those with no inputs) to cover cases with inputs as well as internal signals.

A logic element implements one signal of the control circuit. It is a logic circuit with one output and one or more inputs and is allowed to have memory, i.e. it may have internal feedback wires. The delay model used is one in which a logic gate can be modeled as an infinitely fast combinational circuit followed by a delay (of unbounded value) at the output, and wires with no delays. In practice, these assumptions hold; long wires which are highly capacitive can be modeled explicitly as delay circuits if necessary.

Definition 4.4 (Network Functions) *Let $\Phi_J = \langle S, T, \delta, s_0 \rangle$ be a state graph defined on J, a set of signals of a network. Then $f : S \times J \rightarrow \{0,1\}$ is a partial function called the* network function, *defined as follows:*

(a) *For $j \in J$, $s \in S$, $f(s,j)$ is called the* implied value *of signal j in state s, defined such that:*

$$f(s,j) = \begin{cases} s'(j) & \text{if } \exists s' \in S : s[j_*\rangle s' \\ s(j) & \text{otherwise.} \end{cases}$$

(b) *The set of all implied values for signal j, $f(j) = \{f(s,j) \mid s \in [s_0\rangle\}$ is the* logic function *of signal j.*

(c) *Hence the network function can be considered as a collection of logic functions of all signals in J: $f = \{f(j) \mid j \in J\}$.*

Remarks on Definition 4.4.

1. The implied value of a signal j in state s is given by (a). If there are states s'_1, s'_2 such that $s[t_1\rangle s'_1$ and $s[t_2\rangle s'_2$ for some $t_1, t_2 \in T$, then there are two cases:

- If either t_1 or t_2 is a transition of j, for instance $t_1 = j_*$ and $t_2 \neq j_*$, then $s'_1(j) \neq s'_2(j) = s(j)$. In which case we always choose $s'_1(j)$ as the value for $f(s,j)$, which is the value resulting from the transition of signal j itself.

- If neither of t_1, t_2 is a transition of signal j then $s'_1(j) = s'_2(j) = s(j)$ and the implied value of j in state s is the same: $f(s, j) = s(j)$.

2. Even though network functions are defined for all signals, only those for non-input signals are necessary for the logic implementation of the networks. This is because only non-input transitions in T_{NI} are generated by logic elements in the networks. If $\delta(s, t)$ is defined for some state $s \in S$ and signal transition $t \in T_{NI}$ then the occurrence of t is caused by the logic elements whenever the network gets into state s. Furthermore, since the firing of t depends on the delays of the logic elements, it cannot be controlled externally. However, if $t \in T_I$ then whenever the network is in state s, it waits for the occurrence of input transition t which is caused by the environment. The means for the environment to detect that the network is in state s is through a communication protocol between the environment and the network. The logic functions for input signals only indicate how external transitions from the environment should interact with the logic network according to the specification.

3. The logic function $f(j)$ of signal j is a Boolean function in $|J|$ variables describing the logic element (whose output is) j. Logic element j may have fewer inputs than $|J|$. This fact does not invalidate our model, for we simply choose $f(j)$ as a function that does not explicitly depend upon all of its variables. This can be accomplished by decomposing the state graph to minimize the interdependence between variables which appear as output and input of a logic element. A state graphs obtained from a well-formed STG can be decomposed in a straightforward manner using the causal relation R defined on the STG. This technique yields very efficient implementation of networks and will be dicussed subsequently in Chapter 6.

4.1.4 An Example

We have presented the syntax of STG, its semantics in terms of state graphs and the network function which describes the logic implementation of the circuit. In this example, we show a STG specification of a C-element, its state graph and network function. Fig. 4.1a shows a circuit with inputs a, b and output c. Thus, its set of signals is $J = \{a, b, c\}$, and $J_I = \{a, b\}$ and $J_O = \{c\}$. The behavior of this circuit is described by the STG in Fig. 4.1b. Since its corresponding uninterpreted net is a marked graph, no places are drawn

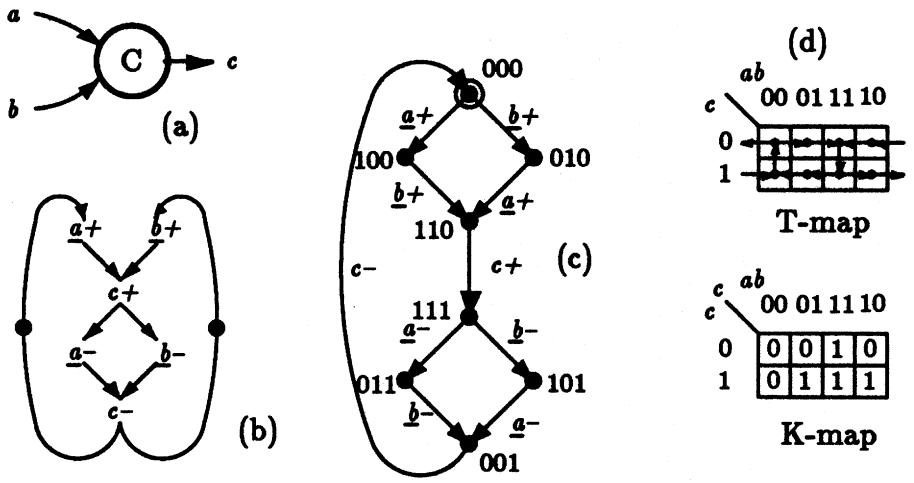


Figure 4.1: (a) A C-element, (b) A STG specification and (c) its state graph. (c) The transition map and K-map.

explicitly; they are uniquely determined by their input and output transitions. The arcs constitute the causal relation R. In the initial marking M_0, places $\langle c_-, a_+ \rangle$ and $\langle c_-, b_+ \rangle$ are marked. The set of transition sequences specified by this STG is given by the regular expression $(a_+c_+a_-c_- \| b_+c_+b_-c_-)^*$, which can be simplified to $((a_+ \| b_+)c_+(a_- \| b_-)c_-)^*$. The corresponding state graph is given in Fig. 4.1c. The initial state s_0 is such that $s_0[a_+\rangle$ and $s_0[b_+\rangle$.

In a state graph, the logic equation for every non-input signal j is determined from the set of implied values of j in all states: $\{ f(s,j) \mid s \in [s_0\rangle \}$. This can be done conveniently by transferring the state graph to a type of Karnaugh maps (K-map) called *transition map* (T-map), such that every state of the state graph has a corresponding square in the K-map, and state transitions are indicated by arcs between squares. Then for each signal j to be determined, the T-map is converted to a K-map for j as follows: every square in the T-map (corresponding to a state s) is entered with the implied value of signal j in state s, $f(s,j)$. The logic equation for j can be readily determined from the resulting K-map. For example, the logic equation for signal c is determined from the state graph in Fig. 4.1c by transfering it to a T-map and K-map (Fig 4.1d) from which, $c = a.b + c(a + b)$. This is precisely the logic function of a C-element.

Note the intimate relationship between a state graph and its logic implementation: states in a state graphs are simply collections of values of signals in a circuit, and transitions

simply transitions of signals in the same circuit. In fact, one can construct a state graph for any circuit. However, we are only interested in circuits whose state graphs satisfy the properties of liveness and persistency. In the next chapter we will study these properties of state graphs and determine their characterizations in STGs.

4.2 Obtaining state graphs from STGs

Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a STG. Then Σ denotes its *uninterpreted* net, also represented by $\langle P, T, F, M_0 \rangle$ but T is given no particular interpretation. Since Σ is a LSFC net its equivalent FA, denoted by Φ, can be determined using the construction algorithm discussed in Chapter 3. The state graph of Σ_J, denoted by Φ_J, can then be obtained by assigning binary values to states of Φ.

Let $\Phi = \langle S', T, \delta', s'_0 \rangle$ denote the above equivalent FA. Then the state graph $\Phi_J = \langle S, T, \delta, s_0 \rangle$ can be obtained by interpreting states in S' as binary vectors representing values of signals in J. This interpretation is the state assignment process, which can be described by a partial function $\alpha : S' \rightarrow S$, called the *state-assignment* function (Fig. 4.2). This function is defined such that

- *Every state s' in Φ maps to a binary state s in Φ_J:* $\forall s' \in [s'_0\rangle \subseteq S' \; \exists s \in S : \alpha(s') = s$.
- *Every transition $\delta'(s'_1, t) = s'_2$ in Φ maps to $\delta(s_1, t) = s_2$ in Φ_J, where $s_1 = \alpha(s'_1)$, $s_2 = \alpha(s'_2)$.*

A state assignment such that δ satisfies the condition stated in Definition 4.2 is called *consistent.* The essential idea is that in a state graph Φ_J, if $s[t\rangle s'$ then according to the physical behavior of digital circuits: if $t = j_+$ then $s(j) = 0$ and $s'(j) = 1$; if $t = j_-$ then $s(j) = 1$ and $s'(j) = 0$. The triple $\langle s, t, s' \rangle$ is said to be *consistent.* Formally, a consistent state assignment for a state graph can be defined as follows.

Definition 4.5 (Consistent state assignment) *Let $\Phi = \langle S', T, \delta', s'_0 \rangle$ be an uninterpreted FA and $\Phi_J = \langle S, T, \delta, s_0 \rangle$ be a state graph obtained from Φ. Then Φ_J has a consistent state assignment iff there exists a state assignment function $\alpha : S' \rightarrow S$ satisfying*

$$\forall t \in T, \forall s'_1, s'_2 \in S' : \quad \textit{whenever } s'_1[t\rangle s'_2 \textit{ then } \langle \alpha(s'_1), t, \alpha(s'_2) \rangle \textit{ is consistent.}$$

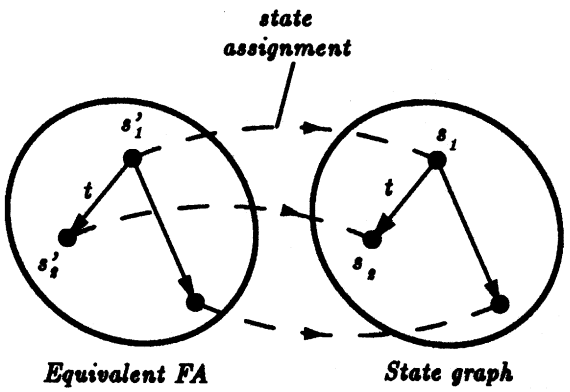


Figure 4.2: The state assignment function α maps a state in the FA to a binary state in the state graph.

Since the transition function δ is defined such that if $\delta(s_1, t) = s_2$ then s_1 and s_2 are adjacent states, it immediately follows that in a state graph with a consistent assignment, every cycle (a path starting from some state and ending at itself) must contain equal numbers of positive and negative instances of any signal transition and they must alternate. Hence, we have the following

Lemma 4.6 *Let $\Phi_J = \langle S, T, \delta, s_0 \rangle$ be a state graph. Then Φ_J has a consistent state assignment iff for every state $s \in S$ and for every firing sequence $\sigma \in T^*$ such that $s[\sigma\rangle s$:*

$$\forall t \in T \begin{cases} \#(\sigma\lceil t) = \#(\sigma\lceil \bar{t}) = k \text{ and} \\ \text{either } \sigma\lceil\{t, \bar{t}\} = (t\bar{t})^k \text{ or } \sigma\lceil\{t, \bar{t}\} = (\bar{t}t)^k, \end{cases}$$

for some integer $k \geq 0$.

In case a transition t does not appear in a sequence σ then $k = 0$ and the above condition holds vacuously. It turns out that there is a direct way to ensure consistent state assignment from STGs. Below, we state the conditions on a STG in order for its state graph to have a consistent state assignment. First we consider the subclass of STGs corresponding to live-safe marked graphs.

Lemma 4.7 *Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a STG whose uninterpreted net is a live-safe marked graph. Then its state graph Φ_J has a consistent state assignment iff every pair of transitions $t, \bar{t} \in T$ is ordered: $\{t, \bar{t}\} \in$* **li**.

Fig. 4.3a shows a STG with consistent state assignment; Fig. 4.3b is a STG with no consistent state assignment because there are transitions b_+, b_- which are concurrent; its state graph contains a state s with $s[b_+\rangle$ and $s[b_-\rangle$ and hence no value of $s(b)$ can be chosen such that both b_+ and b_- are consistent for s.

Proof. ($\Rightarrow$) $\exists t \in T : \{t, \bar{t}\} \notin \mathbf{li} \Rightarrow \Phi_J$ does not have a consistent assignment: Since Σ_J is a marked graph, $\{t, \bar{t}\} \notin \mathbf{li} \Rightarrow \{t, \bar{t}\} \in \mathbf{co}$. This implies that $\exists s \in [s_0\rangle : s[t\rangle s' \wedge s[\bar{t}\rangle s''$, for some $s', s'' \in S$. Obviously, if $\langle s, t, s'\rangle$ is consistent then $\langle s, \bar{t}, s''\rangle$ cannot be, and vice versa. Hence Φ_J cannot have a consistent state assignment.

($\Leftarrow$) $\forall t \in T : \{t, \bar{t}\} \in \mathbf{li} \Rightarrow \Phi_J$ has a consistent assignment: According to Theorem 3.8, the set of firing sequences of Σ_J can be obtained by veawing those of its SM-components. Since Σ_J is a marked graph, these SM-components are simple cycles. Since $t, \bar{t}$ are ordered, they must belong to some simple cycle Ω in N_Σ. Hence the firing sequence defined by Ω must have the following form (given in terms of regular expressions): either $(\ldots t \ldots \bar{t} \ldots)^*$ or $(\ldots \bar{t} \ldots t \ldots)^*$, depending on which place in Ω contains a token in the initial marking.

Choose any firing sequence $\sigma \in T^*$ of Σ_J such that $s_1[\sigma\rangle s_1$ for some $s_1 \in [s_0\rangle$. Since Σ_J is a marked graph, Theorem 3.4 shows that σ must contain an instance of every transition in T. Furthermore, as discussed in the above paragraph, for every pair of transitions $t, \bar{t}$, either $\sigma\lceil\{t, \bar{t}\} = t\bar{t}$ or $\sigma\lceil\{t, \bar{t}\} = \bar{t}t$. Hence according to Lemma 4.7, Φ_J has a consistent state assignment. ∎

The above result can be generalized to the class of STGs corresponding to LSFC nets.

Theorem 4.8 *Let Σ_J be a STG and Φ_J its state graph. Then Φ_J has a consistent state assignment iff every pair of transitions $t, \bar{t}$ is ordered in Σ_J.*

Fig. 4.3c shows another STG with consistent state assignment which specifies *input choices*; its uninterpreted net is a LSFC net. Note the requirement that every pair of transitions $\{t, \bar{t}\}$ be ordered in *every* MG-component containing them. Due to this rather stringent requirement for consistent state assignment, the expressive power of STGs for specifying input choices is limited.

Proof. Let $\Sigma_J = \langle P, T, F, M_0\rangle$. According to Theorem 3.6, there exists for each MG-component $N_i = \langle P_i, T_i, F_i\rangle$ a marking $M_i \in [M_0\rangle$ at which N_i operates as a live-safe

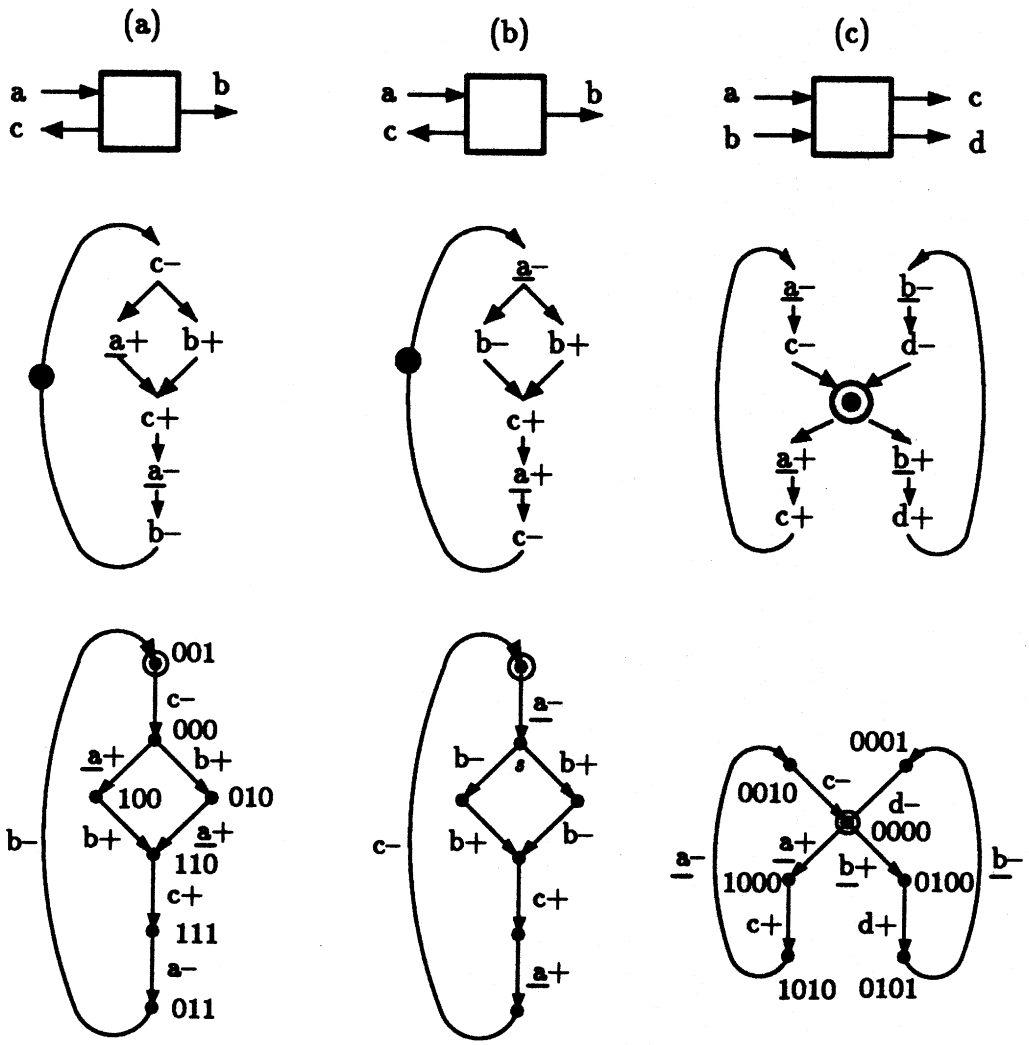


Figure 4.3: (a) A STG with consistent state assignment. (b) A STG without consistent state assignment because b_+ and b_- are concurrent. (c) Another STG specifying input choices which has a consistent state assignment.

marked graph. Hence, the net Σ_J can operate deterministically in one MG-component N_i and switches over to another, N_j say, at some appropriate marking M. Marking M must mark some place p which is a free-choice place (i.e. $|p\cdot| > 1$) such that some transitions $t_i \in T_i,\ t_j \in T_j$ must be output transitions of p.

Since every MG-component N_i can be operated continuously from some marking M_i, the set of firing sequences generated by $\langle N_i, (M_i\lceil P_i)\rangle$ must have an equivalent state graph, which in turn must also be a part of the state graph of Σ_J. Therefore, for the state graph of Σ_J to have a consistent assignment, it is required that the state graph Φ_i of every MG-component $\langle N_i, (M_i\lceil P_i)\rangle$ have a consistent assignment. Using the result from Lemma 4.7 above, we conclude that every pair of signal transitions $t, \bar{t}$ must be ordered in every MG-component N_i to which they belong. It then follows from the definition of ordering that $t, \bar{t}$ are ordered. ∎

4.3 Composition

As stated earlier, the objective of this thesis is to develop techniques for direct synthesis of control circuits from STG specifications. Within this scope, the study of composition of control modules will be touched upon lightly. A great deal of work in different aspects of composition of control modules has been carried out; most relevant to our application is one based on the formalism of trace theory [25,53]. Since we have established the correspondence between net syntax and its underlying trace (firing sequence) semantics, adapting these results to our framework is straightforward. Even though composition is of fundamental importance for validating the correctness of systems constructed from an interconnection of control modules it is outside the scope of our immediate concern.

We advocate the use of STGs for direct synthesis of control modules in a system organization with *distributed control structures*. In these systems, the control section is partitioned into a number of control modules which communicate with one another using a communication discipline such as the *request/acknowledge* protocol. Once the control modules have been identified clearly, their behavior is expressed in STG notations. Every module has its own STG specification which defines its internal behavior with respect to the *interface* with other modules. The interface between a module and its external world constitutes a boundary between a *module* and its *environment*.

Interface behavior. Fig. 4.4a shows a simple case of *module/environment* interface. The module A shown consists of one input *link* $\{I_r, I_a\}$ and one output *link* $\{O_r, O_a\}$, where a link is a pair of *request/acknowledge* signals. We use the convention that for a link L, $\{L_r, L_a\}$ denotes its *request/acknowledge* signal pair. The behavior of module A at the interface is specified by $\{I_{a+} \rightarrow \underline{I_{r-}}, I_{a-} \rightarrow \underline{I_{r+}}\}$ and $\{O_{r+} \rightarrow \underline{O_{a-}}, O_{r-} \rightarrow \underline{O_{a+}}\}$ at the input link I and output link O, respectively; the arrows $\rightarrow$ are members of the causal relation R defined earlier. In general, we have the following rule for specifying the interface for STGs: every transition t of an *input* signal can be caused only by a transition of an *output* signal, u say, and furthermore, there can be exactly one u causing t. More formally, in a STG it is required that *for every* $t \in T_I$, *there exists exactly one* u *such that* $u \rightarrow t$ *and* $u \in T_O$. The reason for allowing only one transition to cause an input transition is that we only attempt to *simulate* the interface behavior by such a temporal constraint; input transitions are actually caused by the environment.

Composition. Given two control modules which are connected together at some link, one can determine the behavior of the composite system by composing the individual behavior of each module. The only type of composition required here is the *concurrent composition* [25]. Informally, the concurrent composition of two STGs involves merging them together by "fusing" transitions in two nets; these fused transitions become internal transitions of the composite net.

An example to illustrate this composition is given in Figures 4.4b and c. Fig.4.4b shows two modules A and B and parts of their STGs which specify the interface behavior. The composition of these STGs produces a new STG (Fig. 4.4c) which no longer corresponds to a free-choice net: transitions r_{1+} and r_{2+} are no longer enabled in the same marking. This illustrates the following important concept: *When two* input *transitions in the STG specification of a module are in* direct conflict *(and therefore represent a nondeterministic choice), they may actually represent external choices which are totally deterministic in the composite net consisting of the module and its environment.* This is exactly the reason why we restrict free-choices in a net to transitions of input signals.

In the following definition of concurrent composition of two STGs, T_{Ii}, T_{Oi} and T_{Ni} denote the subsets of input, output and internal transitions, respectively, of the set of transitions T_i.

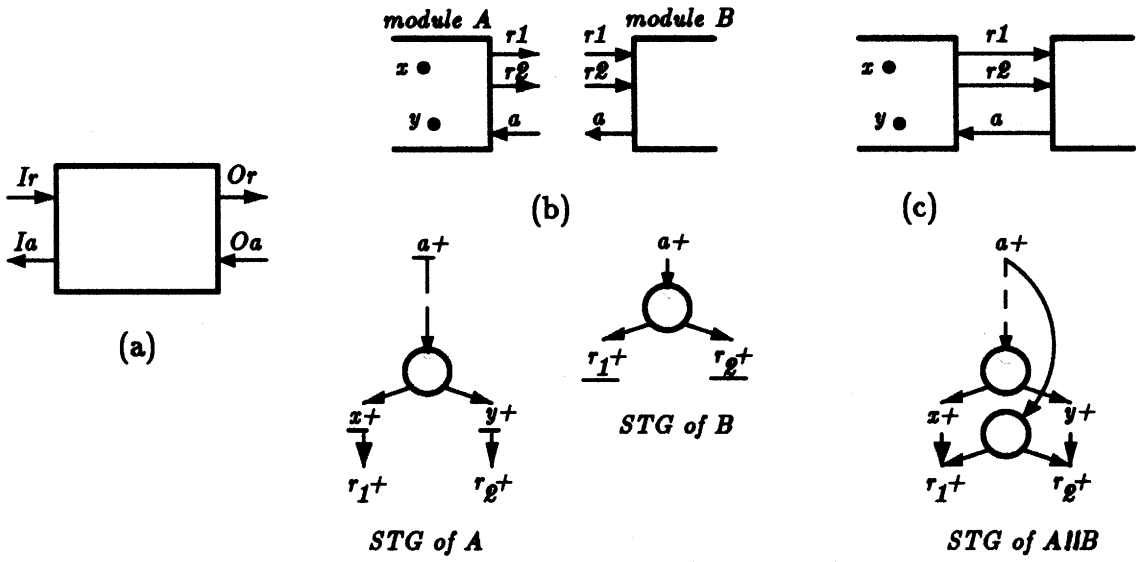


Figure 4.4: (a) A control module with an input link and an output link. (b) Two control modules with connected communation links and (c) their composition.

Definition 4.9 *Let* $\Sigma_{J1} = \langle P_1, T_1, F_1, M_0^1 \rangle$ *and* $\Sigma_{J2} = \langle P_2, T_2, F_2, M_0^2 \rangle$ *be STG specifications of two control modules, with the restriction that*

$$T_{I1} \cap T_{O2} \neq \emptyset,\ T_{I2} \cap T_{O1} \neq \emptyset,\ T_{I1} \cap T_{I2} = T_{O1} \cap T_{O2} = P_1 \cap P_2 = \emptyset.$$

The concurrent composition of Σ_{J1} *and* Σ_{J2} *is denoted by* $\Sigma_J = \Sigma_{J1} \| \Sigma_{J2}$. *Let* $\Sigma_J = \langle P, T, F, M_0 \rangle$, *then it is defined as follows:*

- $J = J_1 \cup J_2,\ J_I = J_{I1} \cup J_{I2} - J_1 \cap J_2,\ J_O = J_{O1} \cup J_{O2} - J_1 \cap J_2,\ J_N = J_{N1} \cup J_{N2} \cup (J_1 \cap J_2)$.
- $T = J \times \{+, -\}$.
- $P = P_1 \cup P_2,\ F = F_1 \cup F_2,\ M_0 = M_0^1 \cup M_0^2$.

Using Theorem 3.8, the set of transition sequences of the composite net is simply the *weave* of the individual sets, provided the latter are of STGs corresponding live-safe marked graphs. Equivalently, the finite automaton of the composite net is the *weave* of those of the individual nets.

Theorem 4.10 *Let* Σ_{J1}, Σ_{J2} *be STGs whose uninterpreted nets are live-safe marked*

graphs, and Φ_{J1}, Φ_{J2} *be their state graphs. Let* $\Sigma_J = \Sigma_{J1} \| \Sigma_{J2}$ *and* Φ_J *denote its state graph. Then* $\Phi_J = \Phi_{J1} \| \Phi_{J2}$, *the weave of two state machines given by Def. 3.11.*

Note that this result does not generalize to composition of STGs whose uninterpreted nets are LSFC nets because, as we have illustrated through Fig. 4.4, the composition of two LSFC nets results in a net which may no longer be free-choice. Also, the above theorem does not guarantee that the composite net Σ_J is live-safe even if Σ_{J1} and Σ_{J2} are live-safe. It may be the case that when two nets are composed, a new simple cycle which contains no token is created in the composite net, leading to a deadlock. However, the theorem does provide a method for verifying whether the composite net is live: Σ_J is live if the weave of none of the firing sequences results in an empty set.

Chapter 5

Properties of State Graphs

In this chapter, we further explore two important properties of state graphs called *liveness* and *persistency*. They are important because they correspond to the properties of deadlock-free and hazard-free in the circuit implementations of state graphs. Since we have established the equivalence between STGs and a class of state graphs, we can characterize these properties in terms of STGs; these will appear as *syntactic* conditions on STGs. Liveness will be considered in Section 1 as it is the simpler to derive of the two properties. Section 2 discusses persistency and its corresponding characterization in STGs. Section 3 describes a problem related to state asssignment and a remedy for this problem.

5.1 Liveness

As discussed in Chapter 2, we will be using a rather restrictive notion of liveness for state graphs. Simply a state graph is live iff it is strongly connected, and each transition is enabled in some state of the state graph. This implies that every state reachable from the initial state of the system is reproducible, i.e. the set of states form an equivalent class in which every two states are mutually reachable. This notion of liveness implies that every transition in the circuit can be enabled infinitely often because it is enabled in at least one state which can be reached from any other state of the circuit.

Definition 5.1 (Liveness of state graphs) *A state graph defined on a set of signal J, $\Phi_J = \langle S, T, \delta, s_0 \rangle$, is* live *iff it is strongly connected and for each $t \in T$, there exists $s \in [s_0\rangle$*

such that $s[t\rangle$.

In state graphs, states and transitions are intimately related, as they represent *binary values* and *transitions* of the same set of signals of a control circuit, respectively. In Section 4.2, we discussed a property of state graphs which requires that every triple $\langle s, t, s' \rangle$, where $s[t\rangle s'$, be consistent. A live state graph must be strongly connected and must have a consistent state assignment.

Based on this definition, the condition on a STG such that its state graph is live can be derived easily. A STG satisfying this *liveness condition* will simply be called *live*. Let Σ_J denote a STG and Σ its uninterpreted net. Then the state graph Φ_J of Σ_J is obtained by performing state assignment on the finite automaton Φ derived from the net Σ. For the state graph Φ_J to be live, it must be strongly connected and have a consistent state assignment. Φ_J is strongly connected iff Σ is a live-safe net; furthermore, because the largest class of nets considered is FC nets, Σ has to be a LSFC net.

The preceding argument simply states that a STG is live iff it is well-formed and its state graph has a consistent state assignment. This establishes the following theorem

Theorem 5.2 (Liveness condition for STGs) *Let Σ_J be a STG and Σ its uninterpreted net. Then Σ_J is live iff*

- *Σ is a live-safe free-choice net, and*
- *In Σ_J, every pair of transitions $t, \bar{t}$ is ordered. (This is precisely the condition for the state graph of Σ_J to have a consistent state assignment, as stated in Theorem 4.8.)*

5.2 Persistency

Persistency one of the most important properties of state graphs, as it is *the* essential property of speed-independent circuits. In Section 2.3, we have briefly discussed this property and its equivalent characterization in STGs. In this section, we will go into detail of how this equivalence is established.

First, we define the concepts of ***enabling*** and ***disabling*** between transitions in uninterpreted finite automata. These general concepts apply to all interpreted FA, including reachability graphs and state graphs. Let $\langle S, T, \delta, s_0 \rangle$ be a FA, where S denotes a set of states and T a set of transitions, both with no particular interpretation. Also let $T_E(s) = \{t \mid s[t\rangle\}$ denote the set of transitions enabled in a state $s \in S$.

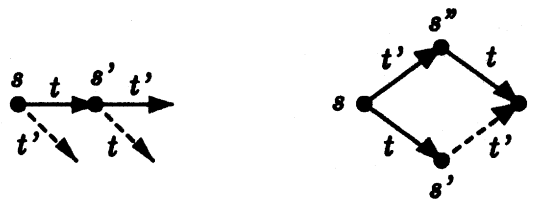
- In state s, t ***enables*** t' (denoted as tEt') iff $\exists s' \in S : s[t\rangle s'[t'\rangle \wedge t' \notin T_E(s) \wedge t \notin T_E(s')$ (Fig. 5.1a).
- In state s, t ***disables*** t' (denoted as tDt') iff $\exists s' \in S : s[t\rangle s' \wedge s[t'\rangle \wedge t' \notin T_E(s')$ (Fig. 5.1b).

Simply, tEt' in s means that the occurrence of t in state s brings the system to another state s' in which t' is enabled; tDt' in s means that the occurrence of t in state s—in which t' is also enabled—brings the system to another state s' in which t' is no longer enabled. We also use $t\bar{E}t'$ and $t\bar{D}t'$ to denote $\neg(tEt')$ and $\neg(tDt')$, respectively. Using this notation, the situation in Fig. 5.1b can be denoted as $tDt' \wedge t'\bar{D}t$.

Generally, in a FA, some transition may have several appearances (instances). Furthermore there are a variety of ways two transitions may interact with each other. For example, there may be instances of transitions t and t' such that tEt' in some state s, and other instances such that tDt' in some other state s'. However, for FA generated from LSFC nets and STGs, we have established their equivalence in Theorem 3.17, which indicates that any situation in a FA has a corresponding characterization in a net. For example, if tRt' (where R is the causal relation defined in Chapters 2 and 4) in a LSFC net then in its FA, an equivalent instance of tEt' must exist, and further there can be no instance of tDt'. The above result is formally stated as follows. Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net and $\Phi = \langle S, T, \delta, s_0 \rangle$ its FA. Then for every $t, t' \in T : tRt'$ in $\Sigma \Leftrightarrow (\exists s \in S : tEt'$ in $\Phi)$.

5.2.1 Definition of Persistency

The property of persistency is defined differently for uninterpreted and interpreted FA—the latter being state graphs. The reason is that in state graphs, transitions are divided into those of ***input*** and ***non-input*** signals, whereas no such distinction exists for uninterpreted FA.



(a) *tEt' in s* (b) *tDt' in s*

Figure 5.1: (a) An instance of t enables t' in s. (b) An instance of t disables t' in s. Dotted arcs are used to indicate transitions which *cannot* occur in certain states.

For uninterpreted FA, a transition is said to be ***persistent*** if none of its instances in a finite automaton is ever disabled by any other transition; otherwise, it is ***non-persistent.*** For example, transition t in Fig. 5.1b is persistent, while t' is non-persistent.

For state graphs, the above definition applies only to transitions of ***non-input*** signals; transitions of ***input*** signals are always assumed to be persistent. This assumption is based on a property of state graphs called the ***external persistency property***, the justification for which will be given shortly.

Definition 5.3 (Persistency in State Graphs) *In a state graph $\Phi_J = \langle S, T, \delta, s_0 \rangle$:*

- *Every $t \in T_I$ is* persistent.
- *$t \in T_{NI}$ is* persistent *iff* $\forall s, s' \in S, \forall t' \in T : \; s[t\rangle \wedge s[t'\rangle s' \Rightarrow s'[t\rangle$*; otherwise, t is* non-persistent.
- *Φ_J is persistent iff every transition in T is persistent.*

An immediate implication of the external persistency property is that it permits the specification of input choices in a state graph. Fig. 5.2b shows an example with an input choice between transitions a_+ and b_+, the corresponding circuit is shown in Fig. 5.2a.

Justification for the External Persistency Property. This intuitive property indeed has a logical justification which is based on the relationship between states and transitions in a state graph. In Fig. 5.2b, states are vectors containing the binary values of signals in $\langle a, b, c, d, e, f \rangle$. From state $s_1 = 000000$, the firing of transition c_+ brings the circuit to state

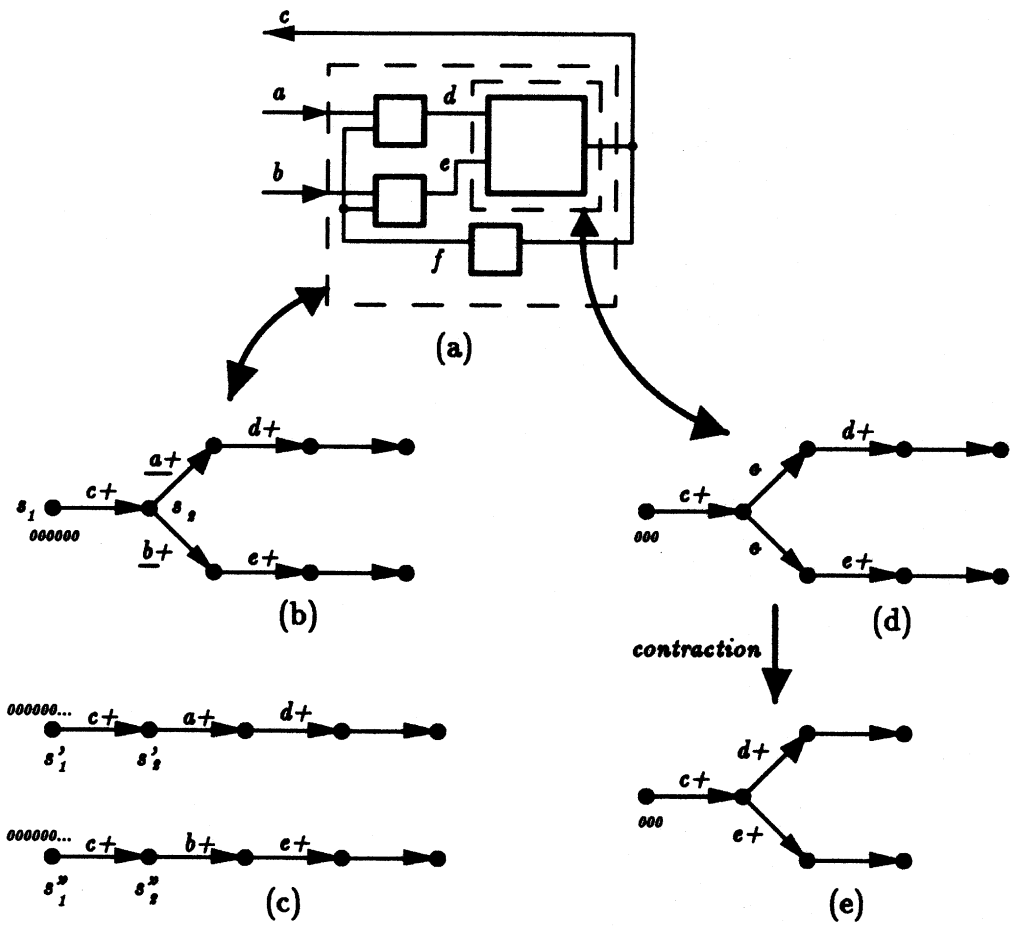


Figure 5.2: (a) A control circuit with an input choice involving signal a, b and (b) part of its state graph. (c) Part of the state graph of the circuit, taken together with its environment. Contraction of the state graph in (b) involves: (d) removing unwanted transitions a_+, b_+, f_+ and (e) contract the state graph.

$s_2 = 001000$, in which both a_+ and b_+ are enabled. Since a and b are external signal, it can be assumed that the environment has additional knowledge to deterministically choose only one of their transitions to fire. This assumption is justifiable as the environment must have enough information to recognize that a_+ and b_+ are not in really enabled in the same state, even though this is how it appears to the circuit. That is, if the circuit and the environment are taken together as a system, then additional signals must exist in the environment such that states s_1 and s_2 can be "split" into $\{s_1', s_1''\}$ and $\{s_2', s_2''\}$, as shown in Fig. 5.2c. Thus, in the more complete system consisting of the original one and its environment, a_+ and b_+ are in fact not enabled in the same state as it appeared in the original one.

This fundamental relationship between states and transitions in a state graph is also the basis for a method of decomposition called *contraction*. It rests on the same principle which relates states and transitions, but instead of considering a system together with its environment, a subcomponent of the system is isolated and its behavior extracted from that of the original state graph. For example, if one is interested in the component consisting only of signals $\{c, d, e\}$ of the circuit in Fig. 5.2a, then a state graph describing its behavior can be derived from in the state graph in Fig. 5.2b as follows. Since other signals $\{a, b, f\}$ do not belong to this subcomponent, their transitions can be removed from the state graph; this is done by replacing them with ϵ (the silent transition) as indicated in Fig. 5.2d. Now each state in the resulting state graph is reduced to a vector of $\langle c, d, e\rangle$. The result is that states connected by ϵ-transitions will have identical binary representations, because these ϵ are transitions of signals which do not belong to this subcomponent. Therefore, they can be collapsed together into one state, as indicated in Fig. 5.2e. In this state graph of the subcomponent, d_+ and e_+ will appear as if they represent an *external choice*, because they are enabled in the same state. The external persistency property dictates that they are persistent, as it is indeed the case. The subject of decomposition using contraction is an important aspect of our synthesis approach. It is discussed in further detail in Chapter 6.

In summary, the external persistency property has two important consequences: (i) transitions of input signals to a control circuit *module* can always be assumed to be persistent, and (ii) input signals to a *logic element* within a module can be assumed likewise.

In the design of self-timed control circuit, the notion of persistency is fundamental as it is the essential property of speed-independent circuits. Chapter 4 shows that from a state

graph, one can define a network function which gives a logic implementation of a digital circuit. This leads to the definition of *speed-independence.*

Definition 5.4 (Speed-independence) *Let $\Phi_J = \langle S, T, \delta, s_0 \rangle$ be a state graph defined over a set of signals J, and $f : S \times J \rightarrow \{0,1\}$ the network function derived from Φ_J. Then the logic circuit which implements f is* speed-independent *iff every transition $t \in T$ is persistent.*

Thus one important consideration in specifying behaviors of speed-independent circuits is to guarantee that all transitions in a STG specification are persistent.

5.2.2 Characterization of Persistency in STGs

We have defined the property of persistency in state graphs. Below we give its characterization in STGs. First we present this syntactic characterization and then give its justification.

Non-persistency in STGs

Fig. 5.3a illustrates a case with u being non-persistent in a state s; it is disabled by a transition t. Using previous notations, this is written as: $tDu \wedge u\bar{D}t$. The equivalent characterization in STG is given in Fig. 5.3b, in which t, u are concurrent and $\bar{t}$ causes u.[1] As explained in Chapter 2, this non-persistency is the result of the interaction of concurrent signal transitions. Intuitively, the course of action $\bar{t}Ru$ can be implemented by a logic element with input i and output j, where $t = i_*$ and $u = j_*$. Concurrency between u and t implies that while the logic element is reacting to transition $\bar{t}$ of signal i to cause transition u of output j, another transition t of i may be occurring simultaneously at the input of logic element j (Fig. 5.3c). This is commonly known as a race condition in hardware circuits and can lead to malfunction. Thus, we have the following syntactic characterization of

[1] Note the difference between this characterization of non-persistency for STGs and that for LSFC nets. In the latter, a transition t is non-persistent only if it is an output transition of a free-choice place. Furthermore, in a LSFC net, if tDu then uDt and vice versa, because in this case, both t and u must share the same input place.

non-persistency in STGs. Recall that T_{NI} denotes the subset of transitions of *non-input* signals.

Theorem 5.5 (Non-persistency in STGs) *Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a live STG and $\Phi_J = \langle S, T, \delta, s_0 \rangle$ its state graph. Then for $t \in T$ and $u \in T_{NI}$:*

$$(\exists s \in S : tDu \wedge u\bar{D}t) \Leftrightarrow (\bar{t}Ru \wedge \{t, u\} \in \mathbf{co}).$$

The proof of this theorem is based on a property of state graph called *input-causing-output*, which states that i is an input signal to logic element j iff some transition of i enables or disables some transition of j. Let $I(j)$ denote the set of input signals to a logic element whose output is j; $I(j)$ is called the *input set* of j. Formally, the *input-causing-output* property states that:

$$i \in I(j) \Leftrightarrow (i_* E j_*) \vee (i_* D j_*).$$

First, we discuss the proof of the above theorem, then we will give an explanation of this property.

Proof. Let $t = i_*$ and $u = j_*$. ($\Leftarrow$) If there is no interaction due to the causal relationship between transitions of signals i and j then the condition $\{t, u\} \in \mathbf{co}$ implies $\exists s \in S$: $s[u\rangle s''[t\rangle \wedge s[t\rangle s'[u\rangle$. We argue that due to the causal relation $\bar{t}Ru$, the condition $\{t, u\} \in \mathbf{co}$ implies $\exists s \in S$: $s[u\rangle s''[t\rangle \wedge s[t\rangle s'$ but not $s'[u\rangle$; i.e. u cannot be enabled in s' or equivalently, tDu.

According to a result developed earlier in this section, the condition $\bar{t}Ru$ implies that $\bar{t}Eu$. Furthermore according to the *input-causing-output property*, it also implies that i is an input to logic element j. Since the STG is live, $\bar{t}$ and t are ordered and hence t must fire some time after $\bar{t}$. Also, since t and u are concurrent, there must be a state s in which both t and u are enabled. At s, the firing of t must disable u because u are enabled by $\bar{t}$ and the firing of t changes the state which results from the firing of $\bar{t}$.

($\Rightarrow$) Since there exists a state s in which both t and u are enabled, it follows that either they are concurrent or in direct-conflict. If they are in direct-conflict then $t\bar{D}u \wedge u\bar{D}t$. However, since we have $tDu \wedge u\bar{D}t$, this is not the case, hence $\{t, u\} \in \mathbf{co}$. Furthermore, according to the *input-causing-output property*, $tDu \Rightarrow i \in I(j) \Rightarrow$ either tRu or $\bar{t}Ru$. But since both t and u are enabled in state s, it cannot be the case that tRu. Hence it must be that $\bar{t}Ru$. Thus, we have $\{t, u\} \in \mathbf{co} \wedge \bar{t}Ru$. ∎

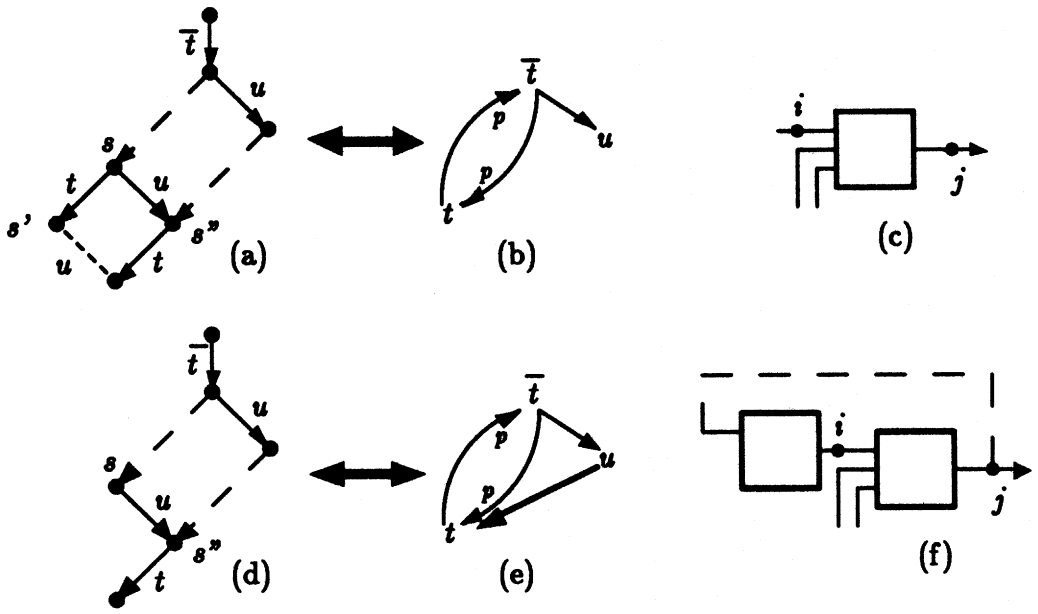


Figure 5.3: Characterization of non-persistency in (a) a state graph, (b) its STG and (c) as hazards in the hardware circuit. The persistency constraint (shown as a bold arrow) in (e) eliminates non-persistency from the state graph (d); harzard is eliminated from the hardware circuit (f). Note that $t = i_*$ and $u = j_*$.

The Input-causing-Output Property. This property reflects the way hardware circuits behave; it is based on the fact that two transitions can enable or disable each other if and only if one is the input to the other. For every signal $j \in J$, let $I(j)$ denote the set of input signals to logic element j. In the following, we give an explanation of this property and show that the input set $I(j)$, as defined above, is indeed equivalent to $\{i \in J \mid i_* R j_*\}$, the set of signals whose transitions cause those of j in a STG. This equivalence is the key to our decomposition technique introduced in Chapter 6.

The *input-causing-output* property states that $\forall i \in J, \forall j \in J_{NI}$ (J is the set of signals): $i \in I(j) \Leftrightarrow \exists s \in S : i_* E j_* \vee i_* D j_*$ in s. Let $t = i_*$ and $u = j_*$. The $\Rightarrow$ direction is obvious: if transitions of i cannot enable or disable those of j then i can be eliminated from the set of input signals to logic element j.

$i \in I(j) \Leftarrow tEu$. Again, consider Fig. 5.1a, in which $s[t\rangle s'[u\rangle$. Note that i is the only signal which changes in going from s to s'. Since u is not enabled in s but it is in s', u must have been disabled due to the transition of signal i. This is possible only if i is an input to logic element j.

$i \in I(j) \Leftarrow tDu$. In Fig. 5.1b, there exist states s, s', s'' such that $s[t\rangle s'$ and $s[u\rangle s''$, but $u \notin T_E(s')$. In state s transition u is enabled, but in s', it is not. Since the only signal which changes going from s to s' is i, u must have been disabled by the transition of signal i. This this is only possible if signal i is an input to the logic element j. ∎

An implication of the above property is the equality $I(j) = \{i \in J \mid i_* R j_*\}$, which can be derived as follows. Recall that $I(j)$ denotes the input set of logic element j.

From the above characterization of non-persistency, $(\exists s \in S : tDu \wedge u\bar{D}t) \Leftrightarrow (\bar{t}Ru \wedge \{t, u\} \in \mathbf{co})$, it is evident that the condition for a transition t to disable another transition u is that $\bar{t}$ *causes* u. Let $t = i_*$ and $u = j_*$, then $tDu \Leftrightarrow i_* D j_*$ and $\bar{t}Ru \Leftrightarrow i_* E j_*$. Hence we have $i_* E j_* \Rightarrow i_* D j_*$. From the input causing output property, we have that since $i_* E j_* \Rightarrow i_* D j_* :\ I(j) = \{i \in J \mid i_* E j_*\} = \{i \in J \mid i_* R j_*\}$.

A syntactic condition for persistency in STGs: the Persistency constraint

There are two mechanisms which give rise to non-persistency. One of which has been characterized above; it arises due to the interaction of concurrent signal transitions which are

parts of the sequencing specification. This *violation of persistency* can always be eliminated by introducing additional ordering constraints into the specification. Such constraints are called *persistency constraints* and are discussed below. The other type of non-persistency is a by-product of the state assignment process, and can only be eliminated by introducing additional bits to differentiate supposedly distinct states, as will be discussed in more detail in the next section.

As illustrated in Fig. 5.3a, transition u is non-persistent as it is caused by $\bar{t}$, but at the same time it is concurrent with t. In order to eliminate non-persistency in this case, we can introduce an ordering constraint between u and t so that they are no longer concurrent; this is shown in Fig. 5.3e as the bold arrow, and it is called a *persistency constraint.* In its state graph (Fig. 5.3d), u becomes persistent because no instance of t can disable it. In the hardware implementation, the introduction of a persistency constraint corresponds to adding a feedback signal from the output j to the input of logic element i, as shown in Fig. 5.3e. We can formalize this as follows.

In Σ_J, let $R \subseteq T \times T$ denote the causal relation. An R-path is defined as a *simple path* $t_1 R t_2 R \ldots R t_n$, for $t_i \in T, 1 \leq i \leq n$ and $t_1 \neq t_n$. Then R^+ denotes the *transitive closure* of R, i.e., $\forall x, y \in T : xR^+y \Leftrightarrow \exists R$-path from x to y. Since Σ_J is strongly connected, R^+ is not a particularly useful concept, for $\forall x, y \in T : xR^+y \wedge yR^+x$. Hence, we define the following *directed* transitive closure $R^\triangleright$:

$\forall x, y \in T (x \neq \bar{y}) : xR^\triangleright y$ *iff there exists an R-path from* x *to* y *which does not contain* $\bar{y}$.

The existence of an R-path from x to y will be depicted graphically as $x \longrightarrow_p y$.

Definition 5.6 (Persistency Constraints) *Let* Σ_J *be a STG as defined earlier. Then* $\forall x \in T$, $\forall y \in T_{NI}$ *such that* xRy*: the R-path such that* $yR^\triangleright\bar{x}$ *is called a* persistency constraint.

It can be seen that the presence of a persistency constraint creates a simple cycle which contains $x, \bar{x}$ and y so that they are ordered. Using in conjunction with the external persistency property, persistency constraints provide the condition on STGs such that their state graphs are persistent. An STG satisfying this condition will be simply called a *persistent* STG.

Theorem 5.7 *Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a STG. Then Σ_J is persistent iff*

$$\forall x \in T,\ \forall y \in T_{NI} :\ xRy \Rightarrow yR^*\bar{x}.$$

5.3 A problem with state-assignment

As mentioned earlier, non-persistency may sometimes arise in a state graph due to state assignments. Nevertheless, we do not incorporate this latter phenomenon into Theorem 5.7 because it is a by-product of state assignments and is by no means related to the required sequencing specification of a STG. In Section 2.5, we have introduced this problem. In this section, we discuss it in more detail.

We generalize the example presented in Section 2.5 to STGs corresponding to marked graphs and LSFC nets. Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LS marked graph and C, C' be cuts in Σ. Then the *interval between* two cuts, denoted by $[C, C']$, is defined as the subset of $P \cup T$ which belongs to all *simple paths* from an element in C to one in C':

$$\begin{aligned}[C, C'] \stackrel{def}{=} \{&x \in P \cup T \mid x \in \text{ path } y_1 y_2 \ldots y_n, \\ &y_1 \in C, y_2 \in C', \\ &y_i \neq y_j, \text{ for } 1 \leq i \neq j \leq n \\ &\text{and } y_i, y_j \in P \cup T\}\end{aligned}$$

An example of an interval between p-cuts C_p and C'_p is shown in Fig. 5.4a, where $C_p = \{\langle x, t_1 \rangle, \langle x, t_3 \rangle\}$ and $C'_p = \{\langle x, t_3 \rangle, \langle \bar{t}_2, t_4 \rangle, \langle t_2, y \rangle\}$. It can be seen that $[C_p, C'_p] \cap T = \{t_1, \bar{t}_1, t_2, \bar{t}_2\}$.

In a STG, a *complementary set* $B \subseteq T$ is defined such that $\forall x \in T : x \in B \Leftrightarrow \bar{x} \in B$. That is, B contains both the rising and falling transitions of a signal. In a live STG corresponding to a marked graph, for every p-cut C_p, $[C_p, C_p]$ forms a complementary set. Then the state assignment problem arises if the following conditions are true in the STG, as illustrated below.

In Fig. 5.4b, markings M and M' correspond to p-cuts C_p and C'_p, respectively. Since $[C_p, C'_p]$ forms a complementary set, from the binary state s corresponding to M, the firing sequence $t_1 \bar{t}_2 \bar{t}_1 t_2$ leads to a binary state s' which is identical to s. Thus in the state graph,

any transition enabled in state s is also enabled in s', and vice versa. In Fig. 5.4b, if t_1 is a transition of a non-input signal, it may oscillate as long as t_3 has not occurred.

On the other hand, the interval $[C_{p1}, C'_{p1}]$ also forms a complementary set, causing the markings M_1 and M'_1 (corresponding to C_{p1} and C'_{p1}, respectively) to have the same binary representation. Hence in the state graph, there is a binary state in which both t_1 and t_4 are enabled. If either t_1 or t_4 is a transition of a non-input signal, non-persitency will result.

In the following lemma, for sets A and B, $A \div B$ denotes their *symmetric difference*: $A \div B = (A - B) \cup (B - A) = A \cup B - A \cap B$.

Lemma 5.8 *Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a STG whose uninterpreted net is a LS marked graph. Then Σ_J has a state assignment problem if there exist distinct p-cuts C_p, C'_p and transition $t \in T_{NI}$ such that*

$$(a) \quad [C_p, C'_p] \text{ forms a complementary set, and}$$
$$(b) \quad t \in (C_p \cdot \div C'_p \cdot) \wedge \cdot t \subseteq C_p \cup C'_p$$

Proof. Let $M, M' \in [M_0\rangle$ denote the marking under which C_p and C'_p are marked, respectively. Consider any firing sequence $\sigma \in T^*$ such that $M[\sigma\rangle M'$, then σ must fire all transitions in $[C_p, C'_p]$. Let s, s' be binary representations of M, M', respectively; s and s' are states in Φ_J. Since $[C_p, C'_p]$ forms a complementary set, if σ contains transitions x then it must also contain $\bar{x}$. Hence $s = s'$ and thus every transition t such that $\cdot t \subseteq C_p$ or $\cdot t \subseteq C'_p$ is enabled in s. There are two cases:

- if transition t belongs to $C_p \cdot \cap C'_p \cdot$ then it is enabled in both markings M and M'. Hence the fact that M and M' have the same binary state cannot cause any problem.
- Therefore, only in case $t \in (C_p \cdot - C'_p \cdot) \cup (C'_p \cdot - C_p \cdot)$ that t is enabled in one marking but not the other. Thus if both markings have the same binary representation, t will be enabled in some state which it is not supposed to. If $t \in T_{NI}$ then Σ_J has a state assignment problem. ∎

This result generalizes directly to STGs which are LSFC nets. As discussed in Chapter 3, Theorem 3.6 states that in a LSFC net, every MG-component can operate independently in its subset of live-safe markings. Hence for LSFC nets, the above conditions for persistency must be satisfied by every MG-component of the net.

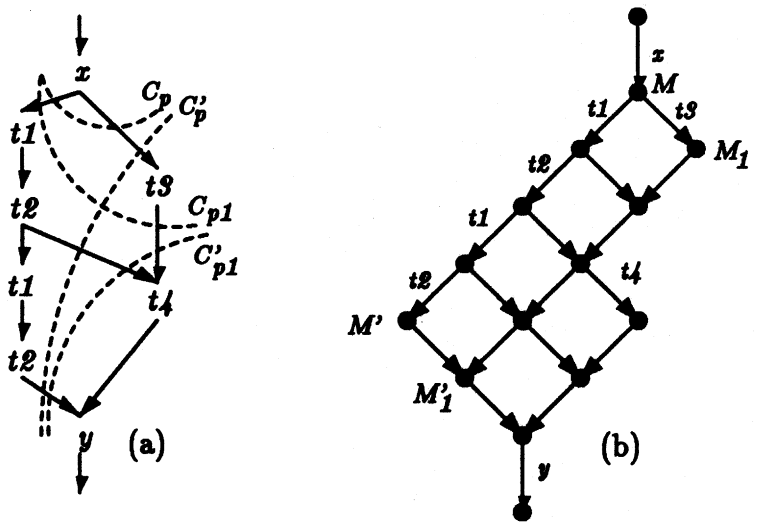


Figure 5.4: (a) A part of a marked graph with p-cuts resulting in non-persistency due to state assignment and (b) its state graph. Note that markings M and M' have the same binary representation.

Theorem 5.9 *Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a STG. Then Σ_J has a state assignment problem if there exist distinct p-cuts C_p, C'_p in some MG-component and a transition $t \in T_{NI}$ such that*

(a) $[C_p, C'_p]$ *forms a complementary set and*
(b) $t \in (C_p \cdot \div C'_p \cdot) \wedge \cdot t \subseteq C_p \cup C'_p$

Chapter 6

Decomposition by Net Contraction

This chapter develops one major result for the structure theory of nets. It discusses a novel method of decomposition of finite automata, using a graph-theoretic technique called *contraction*. We are only concerned with finite automata which are derived from LSFC nets, and for convenience, they will be referred to simply as FA in this chapter. This method does not decompose the FA directly; instead, it contracts the nets to produce smaller ones and then generates FA for these smaller nets.

The purpose of decomposition of FA is to facilitate analysis and synthesis of systems in an effective manner. One specific application of decomposition is in the implementation of control circuits. Efficient implementations can be obtained by decomposing the state graphs to minimize the interaction between variables corresponding to signals of the circuit. For state graphs (which are FA derived from STGs) there is a straightforward method of decomposition based on the *causal relation* in the STGs. STGs and state graphs can be decomposed by performing *contraction* on them to produce a number of contracted graphs; each of which contains the minimum amount of information required for the correct implementation of each signal in the circuit.

As explained earlier, we are mainly interested in the behavior of nets as given by sets of transition sequences. From this viewpoint, an abstraction of net behavior basically involves the removal of unwanted transitions from the sequences. Hence, our abstraction method will involve contracting a net by eliminating unwanted transitions from it.

In Section 1, we describe two contraction algorithms, one for Petri nets and one for finite

automata corresponding to reachability graphs. An example will be given to illustrate the operations involved in these algorithms. In Section 2, we study the conditions under which certain properties of nets are preserved. These include live and safe-ness, and a property called *tr-preserving*, indicating the preservation of the temporal relation in contracted nets. In Section 3, we develop a major result which establish the equivalence between the behavior of a set of contracted nets and the behavior of the original net. In Section 4, these net-theoretic results are applied to STGs to provide a simple method for decomposition of state graphs, from which highly efficient implementations can be obtained.

It is important to note that these results only apply to nets which satisfy all the restrictions of structure theory stated so far in the thesis. To recapitulate, these restrictions on nets consist of the following: *(i) finiteness and safeness (ii) the use of unlabeled transition sequences and (iii) the one-token SM restriction.* The most important of these is the one-token restriction because, as discussed in Chapters 2 and 3, the temporal relation can only be defined for nets satisfying this restriction. In this chapter, all results developed are based on this restriction.

6.1 Contraction Algorithms

6.1.1 Contraction of Petri nets

In this section, we describe contraction algorithms for free-choice nets and their FA. The essential idea in these algorithms is to consider only the subset of transitions of direct interest and ignore the rest. A contracted net contains only transitions of interest, and other transitions are eliminated by performing "local surgery" to remove one transition at a time. It should be stressed that *contraction* is different from *reduction* of nets for obtaining component subnets as described earlier in Chapter 2.

Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net, $T' \subseteq T$ be the subset of transitions of interest. Then Σ/T' denotes the *T'-contracted* net of Σ. In the following contraction algorithm, the set $X = T - T'$ will be eliminated from Σ, resulting in $\Sigma' = \Sigma/T'$. The algorithm eliminates one transition in X at a time and is applied iteratively to the net until all transitions in X are removed. In each iteration, the elimination of a transition $t_e \in X$ only affects places in $\cdot t_e \cup t_e \cdot$ and arcs connected to these places, the rest of the net remains unchanged.

Algorithm 6.1 (Net contraction) *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a net and $T' = T - \{t_e\}$. The T'-contracted net of Σ, $\Sigma' = \langle P', T', F', M_0' \rangle$ is obtained as follows.*

(a) *Collapse input and output places of t_e to create a new set of places P':*
Let $P' = {\cdot t_e} \times t_e{\cdot}$, i.e., $\forall p_i \in {\cdot t_e} \forall p_j \in t_e{\cdot}$: let $p' = \{p_i, p_j\} \in P'$.

(b) *Redefine the flow relation between these new places and the rest of the net:*

$$\forall p_i \in p' (\in P') \quad \textit{if} \quad \exists t \in T' : \langle t, p_i \rangle \in F \quad \textit{then} \quad \langle t, p' \rangle \in F',$$
$$\forall p_j \in p' (\in P') \quad \textit{if} \quad \exists t \in T' : \langle p_j, t \rangle \in F \quad \textit{then} \quad \langle p', t \rangle \in F'.$$

(c) *Replace the remaining part of the net:*
Add $P_1 = P - ({\cdot t_e} \cup t_e{\cdot})$ to P'. Add $F \cap (P_1 \times T' \cup T' \times P_1)$ to F'.

(d) *Determine the new initial marking: $\forall p' \in P' : M_0'(p') = \sum_{p \in p'} M_0(p)$.*

Some examples of net contraction are given in Fig. 6.5, where (a) is a contraction on a part of a marked graph, (b) a state machine and (c) a free-choice net. Note that a contracted net of a free-choice net may no longer satisfy the free-choice axiom; however, it is still behaviorally equivalent to a free-choice net. Even though we trust that the results developed here also apply to these nets which are non-FC (but behaviorally FC, and the like), to avoid net-theoretic difficulties, we will require that contraction be only applied to transitions which do not create non-FC structures in contracted nets.

Notes. After every iteration of the algorithm, the contracted net Σ' must be checked for *place-simple-* and *pure-ness*: If there exist distinct places p_1 and p_2 in Σ' such that $({\cdot p_1} = {\cdot p_2}) \wedge (p_1{\cdot} = p_2{\cdot})$, then it is legitimate to remove either p_1 or p_2 to make Σ' simple. If there exist a transition t and a place p in Σ' such that $p{\cdot} = \{t\}$ and ${\cdot p} = \{t\}$, then the self-loop containing p, t can be eliminated by removing p, thus keeping the contracted net pure. This is also a legitimate operation because p is connected only to t and no other transition; p is often called a *side condition*. On the other hand, place p cannot be eliminated if it has other input or output transitions different from t. In which case the resulting contracted net is impure. Hence, even if the original net Σ is pure, Σ' may not be. Below we provide some restrictions on the contraction algorithm so that these undesirable situations will not occur.

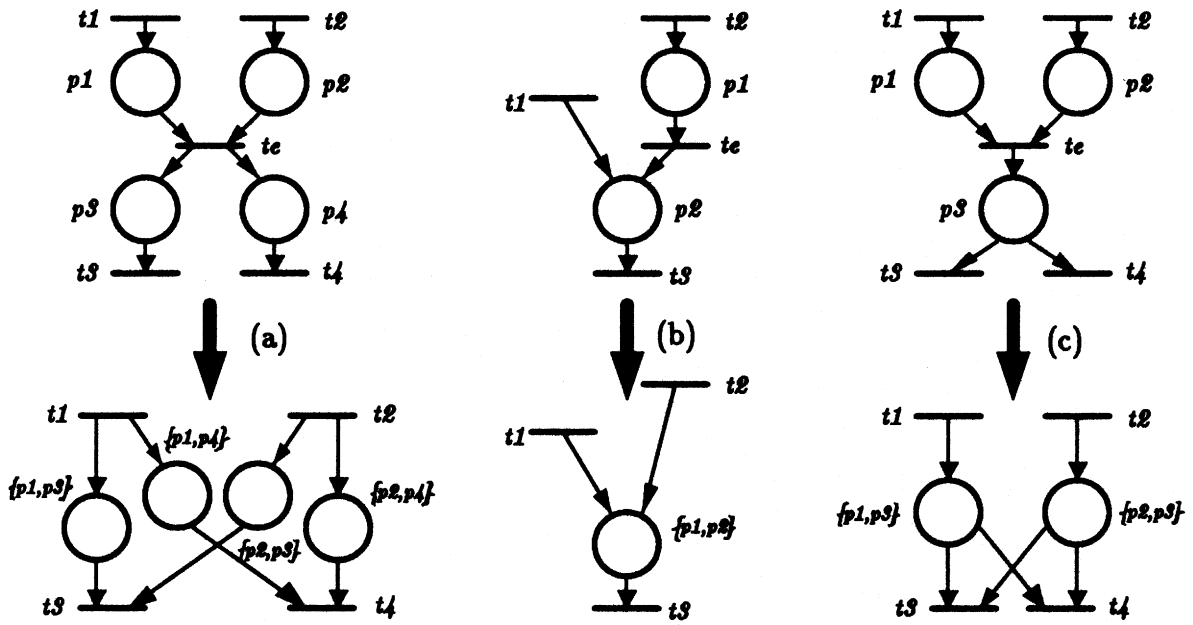


Figure 6.1: Examples of contraction on (a) a marked graph, (b) a state machine and (c) a free-choice net.

Restriction 6.2 (on net contraction) *In the above contraction algorithm, a transition t_e can be eliminated only if it satisfies the following conditions.*

(*a*) *No place $p \in (\cdot t_e \cup t_e \cdot)$ can be an input place of transition $t \in \cdot(\cdot t_e)$ (Fig. 6.2 a):*

$$\forall t \in \cdot(\cdot t_e) : p \in (\cdot t_e \cup t_e \cdot) \Rightarrow p \notin \cdot t,$$

(*b*) *No place $p' \in (\cdot t_e \cup t_e \cdot)$ can be an output place of transition $t' \in (t_e \cdot) \cdot$ (Fig. 6.2b):*

$$\forall t' \in (t_e \cdot) \cdot : p' \in (\cdot t_e \cup t_e \cdot) \Rightarrow p' \notin t' \cdot .$$

It is easy to verify that these situtations create either impurities or deadlocks in contracted nets and hence should not be allowed. In the above algorithm, if there exists a place $p_i \in \cdot t_e$ such that p_i is also an input place of $t \in \cdot(\cdot t_e)$, then since the net is safe, this leads to a deadlock. On the other hand, if there exists a place $p_j \in t_e \cdot$ such that p_j is also an input place of $t \in \cdot(\cdot t_e)$, then the path $tp_it_ep_jt$ exists (where $p_i \in t \cdot \cap \cdot t_e$). After t_e is eliminated, there will be a self-loop $tp't$ where $p' = \{p_i, p_j\}$, resutling in an impure net. As a matter of fact, these situations are so unnatural that it is reasonable to put forth

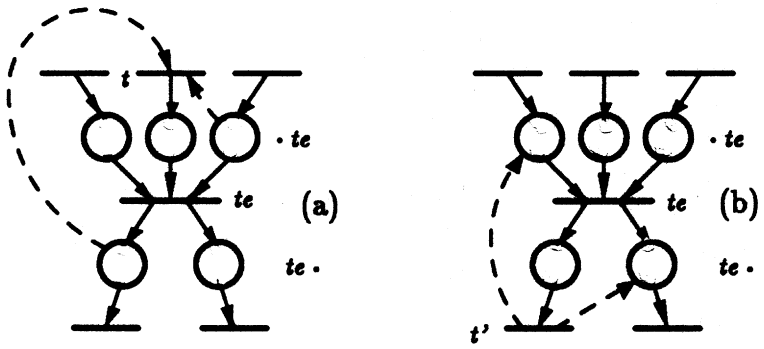


Figure 6.2: Elimination of t_e is disallowed if some input or output place of t_e is also (a) input place of t or (b) output place of t'. The dashed arcs are those causing these violations.

an additional requirement for nets, which is that *all* transitions in a net satisfy the above restriction.

6.1.2 Contraction of Finite Automata

An example has been given in Chapter 5 (in the subsection entitled "Justification for the External Persistency Property") to illustrate the main idea behind contractions of state graphs. Below, we consider contraction in more general terms as applied to uninterpreted FA. One can define a contraction on FA by restricting the set of transitions to a subset of interest and replacing the rest with ϵ, the silent transition. In general, since a FA may have more than one instance of any transition, such an operation on FA may result in nondeterminism in the contracted FA if there is a state with more than one emanating arcs, each labeled with the same transition (including the silent transition ϵ). For the purpose of implementation, we are only interested in systems whose behaviors are described by deterministic finite automata. Hence these nondeterministic FA need to be converted to deterministic ones. Instead of using common method such as the *subset construction* method to produce a deterministic automaton, we use the following two simple rules which are actually properties of the transition function of state graphs. They have been discussed informally before in Chapter 5, and are stated in formal terms below. Let $\Phi = \langle S, T, \delta, s_0, q \rangle$ be a FA, then it is required that

(a) $\forall s, s' \in S, \forall t \in T \cup \{\epsilon\} : \ (s' = \delta(s,t) \wedge t = \epsilon) \ \Leftrightarrow \ s = s'.$

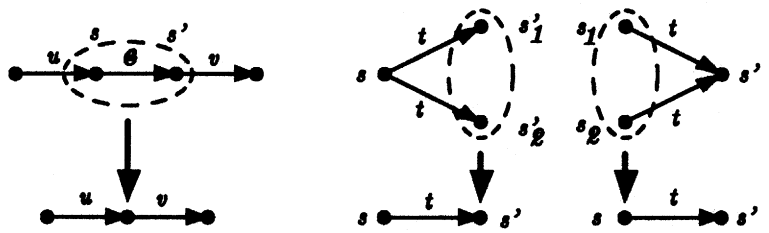


Figure 6.3: Illustration of rules for contraction of FA.

(*b*) $\forall s_1, s_2, s_1', s_2' \in S, \forall t_1, t_2 \in T : s_1' = \delta(s_1, t_1) \wedge s_2' = \delta(s_2, t_2) \wedge (t_1 = t_2) \Rightarrow (s_1 = s_2 \Leftrightarrow s_1' = s_2')$.

Rule (a) states that if two states are connected by a ϵ-transition, then they can be collapsed into one "super-state" (Fig. 6.3a). Rule (b) can be rephrased as two sub-rules: if (i) $s_1 = s_2$ and $t_1 = t_2$ then s_1', s_2' must be the same and can be collapsed into one state (Fig. 6.3b); similarly, if (ii) $s_1' = s_2'$ and $t_1 = t_2$ then s_1, s_2 must be the same and can be collapsed into one state (Fig. 6.3c). The reasoning behind these rules has been described in Chapter 5: silent transitions correspond to transitions of signals not considered in the state of the module of interest and hence, they are eliminated from the state vectors.

In a nondeterministic FA $\langle S, T, \Delta, s_0, q \rangle$ where $\Delta \subseteq S \times (T \cup \{\epsilon\}) \times S$, we say that an ϵ-path exists from state s_1 to s_n, denoted as $s_1[\epsilon \ldots \epsilon\rangle s_n$, if there exists a path $s_1 s_2 \ldots s_n$ such that $\langle s_i, \epsilon, s_{i+1} \rangle \in \Delta$, $1 \leq i < n$. Note that $\forall s \in S : s[\epsilon\rangle s$ and hence there is always a zero-length ϵ-path from any state s to itself. In the following algorithm, let Φ and Φ/T' denote a FA and its T'-contracted FA. The idea of the algorithm is to replace transitions to be removed with ϵ and then collapsed states connected by ϵ-paths into a super-state; nondeterminism can be resolved by applying the above rules; the new initial state will be one which contains the original initial state. In contrast to the contraction algorithm for nets which removes one transition at a time, the algorithm for FA involves the removal of all instances of a transition. The simple reason for this difference is that in a net, each transition in T is required to have exactly one appearance, whereas in a FA, it can appear many times.

Algorithm 6.3 (Contraction of Finite Automata) *Let* $\Phi = \langle S, T, \delta, s_0, q \rangle$ *and* $T' \subseteq T$. *The* T'*-contracted FA of* Φ, $\Phi' = \Phi/T' = \langle S', T', \delta', s_0', q' \rangle$, *is obtained as follows.*

(a) *Relabel unwanted transitions with ϵ: let $\Delta \subseteq S \times (T \cup \{\epsilon\}) \times S$ such that*

$$\forall s, s' \in S, \forall t \in T : \delta(s,t) = s' \Rightarrow \langle s, t\lceil T', s' \rangle \in \Delta.$$

(b) *Collapse into a superstate all states connected with ϵ-paths: $\forall s_1, s_2 \in S$ such that $s_1[\epsilon \ldots \epsilon\rangle s_2$ or $s_2[\epsilon \ldots \epsilon\rangle s_1$, define $s' \in S'$ ($\subseteq \mathcal{P}(S)$, the power set of S) such that $s_1, s_2 \in s'$.*

(c) *Redefine transition relation: Define $\Delta' \subseteq S' \times T' \times S'$ such that $\forall s \in S, \forall t \in T$:*

$$(\exists s_i \in s' \in S' : \langle s, t, s_i \rangle \in \Delta) \Rightarrow \langle s, t, s' \rangle \in \Delta'$$
$$(\exists s_i \in s' \in S' : \langle s_i, t, s \rangle \in \Delta) \Rightarrow \langle s', t, s \rangle \in \Delta'.$$

(d) *Resolve nondeterminism: $\forall t \in T', \forall s_1, s_2, s_1', s_2' \in S'$:*

$$(\langle s_1, t, s_1' \rangle \in \Delta' \wedge \langle s_2, t, s_2' \rangle \in \Delta') \Rightarrow (s_1 = s_2 \Leftrightarrow s_1' = s_2').$$

(e) *The new transition function is obtained as follows. Define $\delta' : S' \times T' \to S'$ such that*

$$\forall s, s' \in S', \forall t \in T' : \langle s, t, s' \rangle \in \Delta' \Rightarrow \delta'(s,t) = s'.$$

The new initial state and the new set of final states are

$$s_0' = s' \in S' \text{ if } s_0 \in s',$$
$$q' = \{s' \in S' \mid s' \cap q \neq \emptyset\}.$$

An example. Fig. 6.4a and Fig. 6.4b are a live-safe marked graph and its state graph, where $T = \{t_0, t_1, \ldots, t_8, t_9\}$. For the sake of clarity, places are not drawn explicitly. By performing T'-contraction, where $T' = T - \{t_2, t_6\}$, the net is contracted to one shown in Fig. 6.4c. The removal of transition t_6 results in the replacement of four old places by four new ones. Similarly, the removal of t_2 causes places $\langle t_1, t_2 \rangle$ and $\langle t_2, t_4 \rangle$ to be collapsed into a new place $\langle t_1, t_4 \rangle$.

The state graph of Fig. 6.4a is given in b. By performing T'-contraction on this state graph, a new one is obtained as shown in Fig. 6.4d. The operation involves replacing t_2 and t_6 with silent transitions ϵ and then collapsing states connected by ϵ-transitions together. Note that the contracted state graph in Fig. 6.4d is also the state graph of the contracted

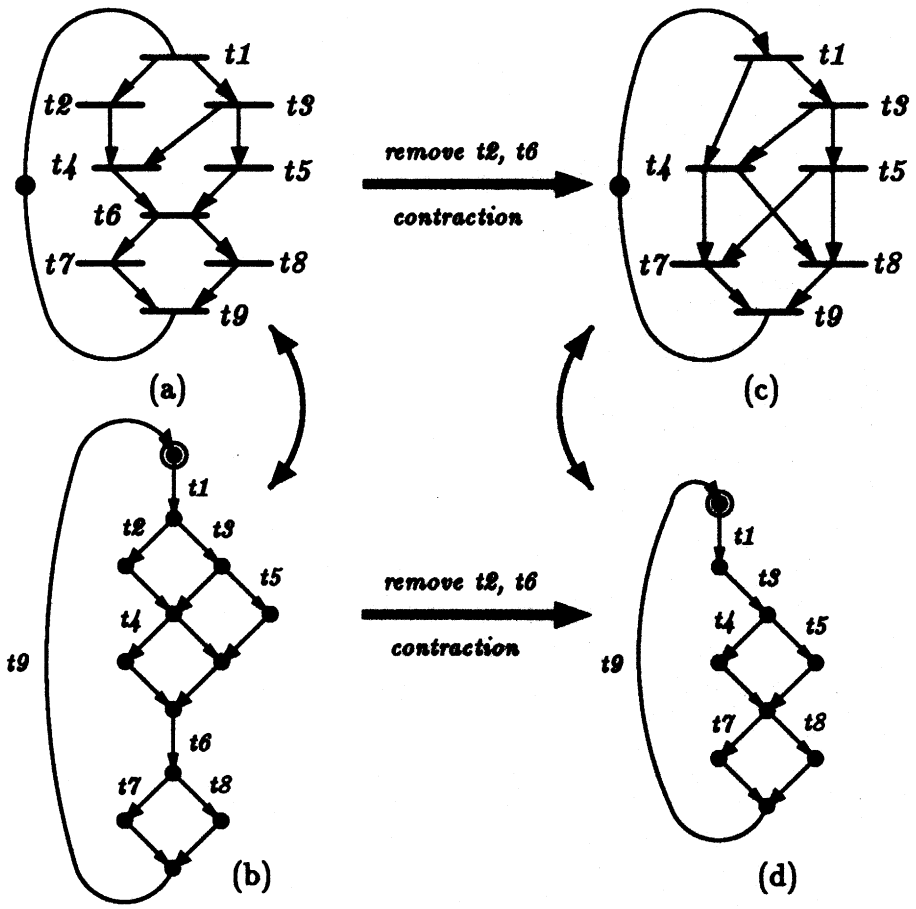


Figure 6.4: (a) Contraction on this FC net yields (c). (b) Contraction of this state graph yields (d). In this example, (d) is also the state graph of (c).

net in Fig. 6.4c. Thus in abstracting the behavior of a net, one can choose either the path a–b–d or a–c–d. In generally, the latter path is computationally more efficient because a net always contains no more transitions than its state graph. In this example, contraction on the net requires the removal of two transitions, while contraction on the state graph requires four (three instances of t_2, one of t_6). In the next section, we will examine the conditions such that both paths yield the same state graph. By satisfying this condition, one can choose the more efficient path a–c–d most of the time.

6.2 Properties of Contraction

Not every subset $T' \subseteq T$ yields a useful or even meaningful contraction on a net. We need to choose subsets with restrictions such that the contracted net preserves certain important properties of the original net. We will study the conditions under which contraction preserves live and safe-ness and the temporal relation on remaining transitions of a contracted net. Due to the level of complexity involved, we will consider separate cases for state machines, marked graphs and lastly, free-choice nets.

In the following, let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net, $\Sigma' = \langle P', T', F', M_0' \rangle = \Sigma/T'$ be the T'-contracted net of Σ. First we introduce the notion of preservation of the temporal relation called *tr-preserving.*

Definition 6.4 *Let Σ be a LSFC net satisfying the one-token SM restriction, and $\Sigma' = \Sigma/T'$ be its T'-contracted net. Let* $\mathbf{tr} = \mathbf{li} \cup \mathbf{co} \cup \mathbf{cf}$ *and* $\mathbf{tr'} = \mathbf{li'} \cup \mathbf{co'} \cup \mathbf{cf'}$ *be the temporal relations in Σ and Σ', respectively. Then the T'-contraction on Σ is* tr-preserving *iff* $\mathbf{li'} \subseteq \mathbf{li} \wedge \mathbf{co'} \subseteq \mathbf{co} \wedge \mathbf{cf'} \subseteq \mathbf{cf}$.

Note that this definition does not require that direct conflicts be preserved. The reason for this is related to the way one defines the boundary of a control circuit module, as explained in Chapter 5.

Below, the condition for tr-preservation for state machines is derived. The removal of transition t_e from a state machine will affect only two places: the input and output places of t_e, because in state machines each transition has exactly one input and one output place (Fig. 6.5).

Lemma 6.5 *Let Σ be a live-safe state machine, Σ' be its T'-contracted net, $T' = T - \{t_e\}$. Let $\{p_i\} = {}^\cdot t_e$ and $\{p_j\} = t_e{}^\cdot$, then the T'-contraction on Σ is tr-preserving iff*

$$|p_i{}^\cdot| = 1 \ \vee \ |{}^\cdot p_j| = 1.$$

This lemma essentially requires that the input place of the removed transition have exactly one output transition, or its output have exactly one input transition. If both places violate

this condition (Fig. 6.5a) then the removal of t_e results in a net in which the temporal relation no longer preserves (Fig. 6.5b).

Proof. First, note that since t_e has only one input and one output place, the removal of t_e results in merging these two places into a new one. Hence if Σ be a live-safe state machine then Σ' is also a live-safe state machine because Σ' is strongly connected and contains exactly one token.

($\Rightarrow$) Suppose that $\exists t_i, t_j \in T : p_i\cdot = \{t_e, t_i\} \wedge \cdot p_j = \{t_e, t_j\}$ (Fig. 6.5), we show that $\mathbf{li}' \not\subset \mathbf{li}$ and $\mathbf{cf}' \not\subset \mathbf{cf}$.

If $\{t_i, t_j\} \in \mathbf{cf}$ then there exists no simple cycle in Σ containing both t_i, t_j. In Σ', p_i and p_j are collapsed into $p' = \{p_i, p_j\}$ and we have $\langle t_j, p'\rangle, \langle p', t_i\rangle \in F'$. Since Σ' is strongly connected, there must be a simple cycle containing both t_i, t_j, i.e. $\{t_i, t_j\} \in \mathbf{li}'$ or $\{t_i, t_j\} \notin \mathbf{cf}'$. Hence $\mathbf{cf}' \not\subset \mathbf{cf}$.

If $\{t_i, t_j\} \in \mathbf{li}$ then there exists a simple cycle in Σ containing both t_i, t_j. There must be a simple path Π' from p_j to p_i because Σ is live. Because t_i and t_j are contained in a simple cycle, there must be another simple path Π from t_i to t_j which does not intersect Π': Π and Π' form such a simple cycle (Fig. 6.5a). Let $t(\neq t_i) \in \Pi$ and $t'(\neq t_j) \in \Pi'$, then it is also true that $\{t, t'\} \in \mathbf{li}$. In Σ', p_i and p_j are collapsed into a new place p' (Fig. 6.5b) and t, t' no longer belong to the same simple cycle. Hence $\{t, t'\} \in \mathbf{cf}'$, or $\{t, t'\} \notin \mathbf{li}'$. So $\mathbf{li}' \not\subset \mathbf{li}$. Note further that a new firing sequence $\ldots t_j t_i \ldots$ is introduced in Σ'.

($\Leftarrow$) It can be verified easily that if either $p_i\cdot = \{t_e\}$ or $\cdot p_j = \{t_e\}$ then the collapsing of p_i and p_j in a new place does not affect the relation between any two transitions. Thus $\mathbf{li}' \subseteq \mathbf{li}$ and $\mathbf{cf}' \subseteq \mathbf{cf}$. ∎

The condition for tr-preservation for marked graphs is more complicated due to the fact that every transition in a marked graph may have more than one input and output places.

Lemma 6.6 *Let Σ be a live-safe marked graph satisfying the one-token SM restriction, Σ' be its T'-contracted net, $T' = T - \{t_e\}$. Then T'-contraction on Σ is tr-preserving iff $\forall t_i \in \cdot(\cdot t_e)$, $\forall t_j \in (t_e\cdot)\cdot$, there exist simple cycles $\Omega_1 = t_i p_i t_e \ldots t_i$ and $\Omega_2 = t_j \ldots t_e p_j t_j$ such that $|\Omega_1 \cap \Omega_2| > 1$ (Fig. 6.6a).*

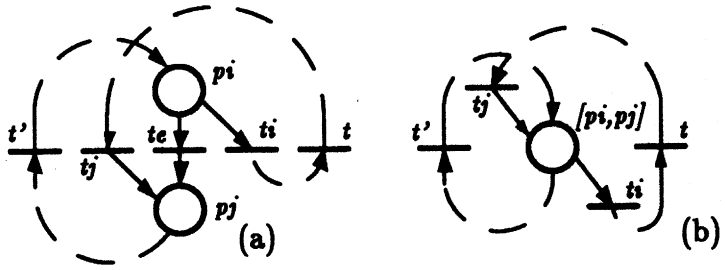


Figure 6.5: Proof of Lemma 6.5. (a) A state machine Σ with $p_i\cdot = \{t_e, t_i\}$ and $\cdot p_j = \{t_e, t_j\}$. (b) The state machine Σ', resulted from collapsing p_i and p_j into p'.

Essentially, the above lemma states that T'-contraction is tr-preserving iff there exists a pair of simple cycles—one containing t_i, t_e, the other t_j, t_e—which intersect at another transition different from t_e. Fig. 6.6a shows a marked graph which violates this condition: the cycles Ω_1 and Ω_2 intersect at only one transition t_e. In which case, the removal of t_e creates a new simple cycle (Fig. 6.6b) which totally alters the temporal relation. Since places in a marked graphs are uniquely identified by their input and output transitions, in order to simplify notations in the following proof, we use the ***causal relation*** R as defined earlier. Recall that $t_iRt_j \Leftrightarrow \exists p \in P : \langle t_i, p\rangle, \langle p, t_j\rangle \in F$.

Proof. For every $t, t' \neq t_e$, if $\{t, t'\} \in \mathbf{li}$ then there exists a simple cycle containing both of them. Then it is easy to see that $\{t, t'\} \in \mathbf{li}'$ because contracting t_e cannot remove any simple cycle from the net. Thus we only need to show that under the condition stated, $\mathbf{co}' \subseteq \mathbf{co}$.

First, note that $C_p = \cdot t_e$ and $C'_p = t_e\cdot$ are p-cuts, and every $t \in \cdot C_p$ is ordered with at least one $t' \in C'_p\cdot$, i.e., $\{t, t'\} \in \mathbf{li}$. Hence, if $\{t_i, t_j\} \in \mathbf{co}$ then $\exists t' \in C'_p\cdot : \{t_i, t'\} \in \mathbf{li}$ and $\exists t \in \cdot C_p : \{t, t_j\} \in \mathbf{li}$. In other words, there exist simple cycles Ω_1, Ω_2 such that $t_i, t' \in \Omega_1$ and $t, t_j \in \Omega_2$; Ω_1, Ω_2 must intersect at t_e (Fig. 6.6a).

($\Rightarrow$) Suppose that $\Omega_1 \cap \Omega_2 = \{t_e\}$. Then in Σ', the removal of t_e introduces t_iRt', t_iRt_j, tRt' and tRt_j (Fig. 6.6b). This introduces a new simple cycle $t_it_j \ldots tt' \ldots t_i$. Hence $\{t_i, t_j\} \notin \mathbf{co}'$. Note further that this cycle always contains two tokens; under the one-token SM restriction, this results in unsafe behaviors.

($\Leftarrow$) Suppose that $\exists u \in T : \Omega_1 \cap \Omega_2 = \{t_e, u\}$. Then in Σ', even though the removal of t_e still introduces t_iRt', t_iRt_j, tRt' and tRt_j, this only creates the non-simple cycle

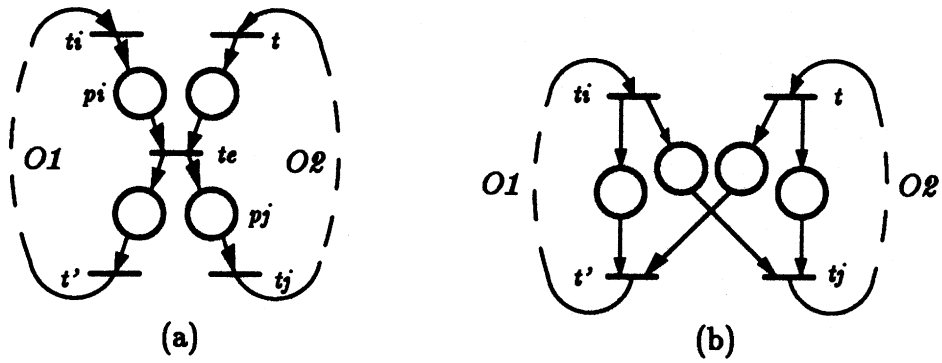


Figure 6.6: Proof of Lemma 6.6 (a) A marked graph Σ before contraction and (b) its contracted graph Σ'.

$t_i t_j \ldots u \ldots t t' \ldots u \ldots t_i$. Hence no new simple cycle is created and $\{t_i, t_j\} \in \mathbf{co}'$. ∎

The following result comes immediately from the proof of the last lemmata.

Corollary 6.7 *Let Σ be a live-safe marked graph (or state machine) satisfying the one-token SM restriction, Σ' be its T'-contracted net, $T' = T - \{t_e\}$. If T'-contraction on Σ is tr-preserving then Σ' is also a live-safe marked graph (state machine).*

Using lemmata 6.5 and 6.6, we can derive the following conditions for tr-preservation for LSFC nets.

Theorem 6.8 *Let Σ be a LSFC net satisfying the one-token SM restriction, Σ' be its T'-contracted net, $T' = T - \{t_e\}$. Then T'-contraction on Σ is tr-preserving iff the following two conditions are true:*

(a) $\forall p_i \in {\cdot t_e} : |p_i \cdot| = 1$ *or* $\forall p_j \in t_e{\cdot} : |\cdot\, p_j| = 1$.

(b) $\forall t_i \in {\cdot}({\cdot t_e})\ \forall t_j \in (t_e{\cdot}){\cdot}$: *there exist simple cycles* $\Omega_1 = t_i p_i t_e \ldots t_i$ *and* $\Omega_2 = t_j \ldots t_e p_j t_j$ *such that* $|\Omega_1 \cap \Omega_2| > 1$.

Proof. (a) The proof for this part is similar to that of Lemma 6.5 for state machine, except that for LSFC net, $|t_e \cdot| \geq 1$ and $|\cdot\, t_e| \geq 1$. According to Lemma 6.5, T'-contraction is *not* tr-preserving iff

$$\exists p_i \in {\cdot t_e}, \exists p_j \in t_e{\cdot} : |p_i \cdot| > 1 \wedge |\cdot\, p_j| > 1.$$

Hence in order for $\mathbf{cf}' \subseteq \mathbf{cf}$, the negation of the above statement must be true; in the case of LSFC nets, this yields (a).

(b) The proof for this part is similar to that of Lemma 6.6 for marked graph. However, for LSFC net if $\{t_i, t_j\} \notin \mathbf{li}$, besides the possibility that $\{t_i, t_j\} \in \mathbf{co}$, one also has to consider the case of $\{t_i, t_j\} \in \mathbf{cf}$. However, in this latter case if condition (a) is satisfied then $\{t_i, t_j\} \in \mathbf{cf}'$. Furthermore, if $\{t_i, t_j\} \in \mathbf{cf}$ then there exist two simple cycles Ω_1 and Ω_2 which intersect at p_i, t_e and p_j; hence $|\Omega_1 \cap \Omega_2| > 1$ and this is consistent with condition (b). ∎

The contraction being tr-preserving implies that local change due to the removal of t_e does not affect the global structure of the net: no new simple cyle can result in Σ' due to contraction, and hence if a simple cycle in N_Σ contains a token, then it still contains a token in $N_{\Sigma'}$. This implies that the contracted net is also live-safe.

The main result of this section is that if Σ is a LSFC net and Σ' is its contracted net such that the contraction is tr-preserving then Σ' is also live-safe. To prove this we first show the following lemma. In the sequel, we will denote the set of firing sequences of a net Σ_i by FS_i, that of a contracted net Σ'_j by FS'_j.

Lemma 6.9 *Let Σ be a LSFC net satisfying the one-token SM restriction, and $\Sigma' = \Sigma/T'$, where $T' = T - \{t_e\}$ and T'-contraction is tr-preserving. Then for every SM-component Σ_1 of Σ, there exists a SM-component Σ'_1 of Σ' such that $\Sigma'_1 = \Sigma_1/T'$ and $FS'_1 = FS_1\lceil T'$, and vice versa.*

Proof. Note that a SM-component comprises of a set of simple cycles which intersect each other at places.

We need to show that (1) for every SM-component Σ_1 of Σ, there exists a SM-component Σ'_1 of Σ_1 such that $\Sigma'_1 = \Sigma_1/T'$ and furthermore, $FS'_1 = FS_1\lceil T'$, and that (2) the contracted net Σ' can have no more SM-components than Σ. If these two facts are true then the lemma is proven.

Consider any SM-component Σ_1 which contains t_e, then in Σ_1 there exists a simple cycle Ω containing t_e. In Ω, let p_i, p_j be the input and output place of t_e, respectively. Since Σ_1 is an SM-component, all transitions connected to p_i and p_j also belong to Σ_1.

Now, eliminate t_e from Σ.

1. When t_e is eliminated from the SM-component Σ_1, every simple cycle containing t_e gets contracted to a new one with t_e removed and places p_i, p_j collapsed into a new place $p' = \{p_i, p_j\}$. All transitions previously connected to p_i and p_j, except t_e, are now connected to p' instead. Hence in the contracted net, every cycle in Σ_1 containing t_e becomes a contracted cycle with t_e removed, and other cycles in Σ_1 which intersect at p_i and p_j will now intersect at p' instead. Thus in the contracted net there is a SM-component Σ_1' which contains all transitions of Σ_1 except t_e. By the definition of net contraction, we have $\Sigma_1' = \Sigma_1/T'$.

Since the contraction is tr-preserving, condition (b) of Lemma 6.8 guarantees that contraction does not create any "new" simple cycle in the contracted net Σ' (new in the sense that it combines previous simple cycles containing concurrent transitions into a new cycle in which these transitions become ordered) and further, any cycle in Σ containing a token will contracts into a cycle containing a token also. Since the contracted SM-component Σ_1' is composed of some of these contracted simple cycles, it is live-safe. Also since condition (a) of Lemma 6.8 is satisfied, $FS_1' = FS_1\lceil T'$.

2. Again, condition (b) of Lemma 6.8 guarantees that no "new" cycle is created in the contracted net Σ'. If such a new cycle were created then a new SM-component must exist in Σ', because every cycle is contained in some SM-component. Since no such new cycle is created by contraction, no new SM-component can result. This implies that the number of SM-components in the contracted net Σ' can be no more than that in the original net Σ. ∎

The above result leads to the following theorem.

Theorem 6.10 *Let Σ be a LSFC net satisfying the one-token SM restriction, and $\Sigma' = \Sigma/T'$ its contracted net,where T'-contraction is tr-preserving. Then Σ' is also live-safe.*

Proof. Since the contraction is tr-preserving, according to Lemma 6.9, every SM-component of Σ' contains one token each and together they cover Σ'. Hence according to Hack's theorem, Σ' is live-safe. ∎

6.3 Decomposition by net contraction

The significance of tr-preserving contractions is that it provides two alternatives for decomposing (by contraction) FA corresponding to reachability graphs of LSFC nets. On one hand, we could perform a desired contraction directly on a FA; on the other hand, we could contract a net and then generate a FA from the contracted net. It turns out the latter approach is almost always more efficient computationally. In case of nets being state machines, contraction on such nets and their FA amounts to the same operation. The reason it is more efficient than contracting a FA directly is that in a net, each transition is allowed to appear exactly once, whereas in a FA, each transition may appear several times. Hence removal of transitions from a net is much easier than removal of transitions from a FA.

Theorem 6.11 *Let Σ be a LSFC net satisfying the one-token SM restriction, Σ' be its T'-contracted net, $T' = T - \{t_e\}$. If T'-contraction on Σ is tr-preserving then*

$$FS(\Sigma') = FS(\Sigma)\lceil T' \quad \textit{or equivalently} \quad \Phi(\Sigma') = \Phi(\Sigma)/T'.$$

Proof. N_Σ and $N_{\Sigma'}$ are identical except for transition t_e, places in $\cdot t_e \cup t_e \cdot$ and arcs connected to these elements (see Fig. 6.7). Let t_i denote a transition in $\cdot(\cdot t_e)$ and t_j one in $(t_e \cdot)\cdot$. Then if $\sigma = \sigma_1 t_i t_e t_j \sigma_2$ is a firing sequence in $FS(\Sigma)$, both sequences $\sigma_1 t_i$ and $t_j \sigma_2$ must belong to $FS(\Sigma)$ and $FS(\Sigma')$ because these sequences do not involve t_e.

Hence, in order to show that $FS(\Sigma') = FS(\Sigma)\lceil T'$, we only need to establish the correspondence between every minimal length sequence $t_i t_e t_j$ in Σ with another $t_i t_j$ in Σ'. Only minimal length sequences need be considered because any transition concurrent with t_i can be fired before it, any transition concurrent with t_j can be fired after it. Also, we need not be concerned with transitions t ($\neq t_e$) which are output transitions of places in $\cdot t_e$ or input transitions of places in $t_e \cdot$ (Fig. 6.7c); it can be easily verified that the removal of t_e does not affect firing sequences containing $t_i t$ or $t t_j$.

Hence our task is to demonstrate the correspondence between $t_i t_e t_j$ in Σ and $t_i t_j$ in Σ'. In Σ, some input places of t_j comes from t_e, some from other transitions; let this latter set of input places be $P_1 (= \cdot t_j - t_e \cdot)$. Due to the FC hypothesis, there are no arcs between any places in P_1 and t_e (Fig. 6.7a). In Σ', since places in P_1 are not connected to t_e by any arcs, they remain the same; hence input places of t_j come from P_1 and $\cdot t_e \times t_e \cdot$ (Fig. 6.7b).

(a) $FS(\Sigma)\lceil T' \subseteq FS(\Sigma')$: We show that for every firing sequence $M_i[t_i\rangle M_e[t_e\rangle M_j[t_j\rangle$ in Σ, there exists $M_i'[t_i\rangle M_j'[t_j\rangle$ in Σ'.

In Σ, $M_i[t_i\rangle M_e[t_e\rangle M_j[t_j\rangle$ is given: At marking M_i all input places of t_i are marked. Since the firing of t_i leads to $M_e[t_e\rangle$, at M_e every place in $\cdot t_e - t_i\cdot$ must have been marked by the concurrent firing of transitions in $\cdot(\cdot t_e - t_i\cdot)$. The firing of t_e leads to $M_j[t_j\rangle$; at M_j all places in $\cdot t_j$ are marked. Since the firing of t_e marks only $t_e\cdot$, at M_e all places in P_1 must have already been marked by the time t_i fires; some of it may have been marked by the firing of some transition concurrent with t_i, or even by t_i itself.

In Σ', let us construct $M_i'[t_i\rangle M_j'[t_j\rangle$: Let Mi', M_j' be markings such that $M_i'[t_i\rangle M_j'$. We know that M_i' exists because it is the same as M_i except for places in $\cdot t_e \times t_e\cdot$. Recall that Restriction 6.2 on net contraction states that none of the places in $\cdot t_e \times t_e\cdot$ can be an input place to t_i. Therefore, if t_i is enabled in M_i, it is also enabled in M_i'. The firing of t_i at M_i' marks $\cdot t_e \times t_e\cdot$ and possibly some of the previously unmarked places of P_1 (as explained above), so that all places of P_1 are marked. Therefore all input places of t_j are marked, leading to M_j'; at this marking, t_j is enabled: $M_j'[t_j\rangle$. We have thus constructed $M_i'[t_i\rangle M_j'[t_j\rangle$.

(b) $FS(\Sigma') \subseteq FS(\Sigma)\lceil T'$: We show that for every firing sequence $M_i'[t_i\rangle M_j'[t_j\rangle$ in Σ' (where $t_i \in \cdot p'$, $t_j \in p'\cdot$ for some $p' \in \cdot t_e \times t_e\cdot$), one can construct $M_i[t_i\rangle M_e[t_e\rangle M_j[t_j\rangle$ in Σ.

In Σ', since $M_j'[t_j\rangle$, at M_j' all places in $\cdot t_j$ are marked. We can break $\cdot t_j$ into P_1 and $P_2 = \cdot t_j - P_1$, the latter being a subset of $\cdot t_e \times t_e\cdot$. Then for every $p' \in P_2$, $M_j'(p') = 1$. Since only P_2 is marked by the firing of t_i, P_1 must have already been marked when t_i fires.

Now in Σ, let us construct $M_i[t_i\rangle M_e[t_e\rangle M_j[t_j\rangle$: Since $p' = \{p_i, p_j\}$ for some $p_i \in \cdot t_e$, $p_j \in t_e\cdot$, in Σ there exists a marking M_j under which p_j is marked. Since p_j is an output place of t_e, M_j must result from the firing of t_e. This implies the existence of another marking M_e which enable t_e. Thus at M_e all places of $\cdot t_e$ are marked, including p_i. Therefore we must have a marking $M_i[t_i\rangle$ at which all places in $\cdot t_e$ except p_i are marked; since $\cdot t_e$ is a p-cut, $\cdot t_e - \{p_i\}$ can be marked before p_i by those transitions concurrent with t_i. Thus we have constructed $M_i[t_i\rangle M_e[t_e\rangle M_j[t_j\rangle$. ∎

The preceding results suggest the following method of abstraction for LSFC nets. Suppose we are interested in the behavior of a subset of transitions $U \subseteq T$ of LSFC net Σ.

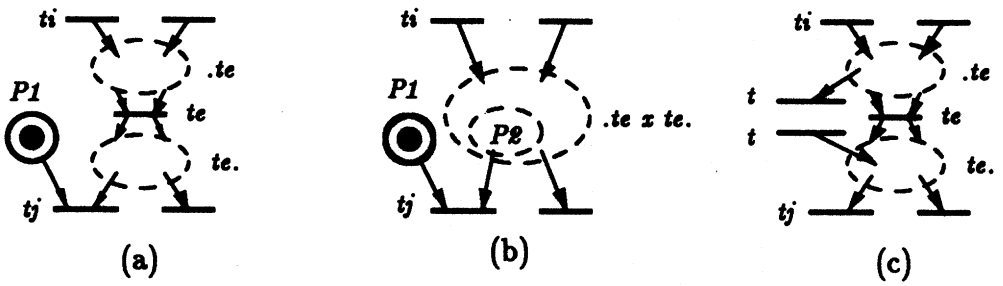


Figure 6.7: Proof of Theorem 6.11 (a) The part of a free-choice net where contraction will occur and (b) its contracted net. (c) Cases which need not be considered.

Then we choose the smallest subset T' such that $U \subseteq T' \subseteq T$ and T'-contraction on Σ is tr-preserving. Then the T'-contracted *FA* of Σ, $\Phi(\Sigma)/T'$, is guaranteed to be identical to the FA of the contracted net $\Sigma' = \Sigma/T'$.

Below, we develop one major result for this chapter concerning the behavioral equivalence of a LSFC net Σ and a set of contracted nets of Σ. We will adopt the following notational convention: (i) for a net Σ, its set of firing sequences is denoted by FS, its causal relation by R; similarly, (ii) for a SM-component Σ_i, they are FS_i and R_i; finally, (iii) for a contracted net Σ'_j, they are FS'_j and R'_j.

Theorem 6.12 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net satisfying the one-token SM restriction, and $\{\Sigma'_1, \Sigma'_2, \ldots, \Sigma'_m\}$ be a set of* contracted *nets of Σ, where $\Sigma'_j = \Sigma/T'_j$, $T'_j \subseteq T$, for $1 \leq j \leq m$.*

For $1 \leq j \leq m$, if (i) T'_j-contraction is tr-preserving and (ii) the union of causal relations R'_j covers R, i.e. $\bigcup_j R'_j \supseteq R$, then

$$FS = FS'_1 \parallel FS'_2 \parallel \ldots \parallel FS'_m.$$

The example in Fig. 6.8 illustrates this theorem. The top leftmost net is a LSFC net Σ which is a marked graph. Also for convenience, we use the graphical abbreviation for STGs to show this net. The rest of the top row contains three SM-components $\{\Sigma_1, \Sigma_2, \Sigma_3\}$, while the rest of the left column contains two contracted nets of Σ: $\{\Sigma'_1, \Sigma'_2\}$. In the contracted nets, solid arcs corresponding to part of the causal relation which are present in the original net Σ; the union of these solid arcs "cover" all arcs in Σ. The dashed arcs in the contracted

nets represent members of the causal relation which have no correspondence in Σ. Also, both contractions are tr-preserving.

The above theorem states that the set of firing sequences of Σ is the identical to one obtained by weaving sets of firing sequences of the contracted nets $\{\Sigma'_1, \Sigma'_2\}$.

The significance of this theorem is that it allows net to be decomposed into smaller nets in any arbitrary way, and each can be used for the purpose of synthesis or analysis. If the conditions stated in the theorem are met, then the behavior of the original net can be obtained by concurrently composing (weaving) the behavior of the smaller nets. Since we have shown that for LSFC nets, every set of firing sequences FS has an equivalent finite automaton Φ, the above theorem implies that

$$\Phi = \Phi'_1 \parallel \Phi'_2 \parallel \ldots \parallel \Phi'_m.$$

Proof of Theorem 6.12. Let $\{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$ be the set of *all* SM-components of Σ. According to Theorem 3.8, the weave of sets of firing sequences of the SM-components gives the set of firing sequences of the net, i.e.,

$$FS = FS_1 \parallel FS_2 \parallel \ldots \parallel FS_n.$$

Hence, every contracted net Σ'_j $(1 \leq j \leq m)$ can be decomposed into n not necessarily distinct SM-components $\{\Sigma'_{j,i}\}$, $1 \leq i \leq n$ such that

$$FS'_j = FS'_{j,1} \parallel FS'_{j,2} \parallel \ldots \parallel FS'_{j,n},$$

where $FS'_{j,i}$ denotes the set of firing sequences of $\Sigma'_{j,i}$. (This is illustrated in Fig. 6.8, where a contracted net in each row of the first column is decomposed into SM-components shown in other columns to its right. Note that even though some of the SM-components are the same, we list them separately to show that they are contracted version of the SM-components on the first row, according to Lemma 6.9.)

As indicated above, according to Lemma 6.9, for every SM-component $\Sigma'_{j,i}$ of the contracted net Σ'_j, there exists a SM-component Σ_i of Σ such that $\Sigma'_{j,i} = \Sigma_i / T'_j$. It is easy to see that if the union of arcs in $\{\Sigma'_j\}$ cover Σ $(\bigcup_j R_j \supseteq R)$, then arcs in $\{\Sigma'_{j,i}\}$ cover the SM-component Σ_i: $\bigcup_j R'_{j,i} \supseteq R_i$. In order to proceed with the rest of the proof, we need the following

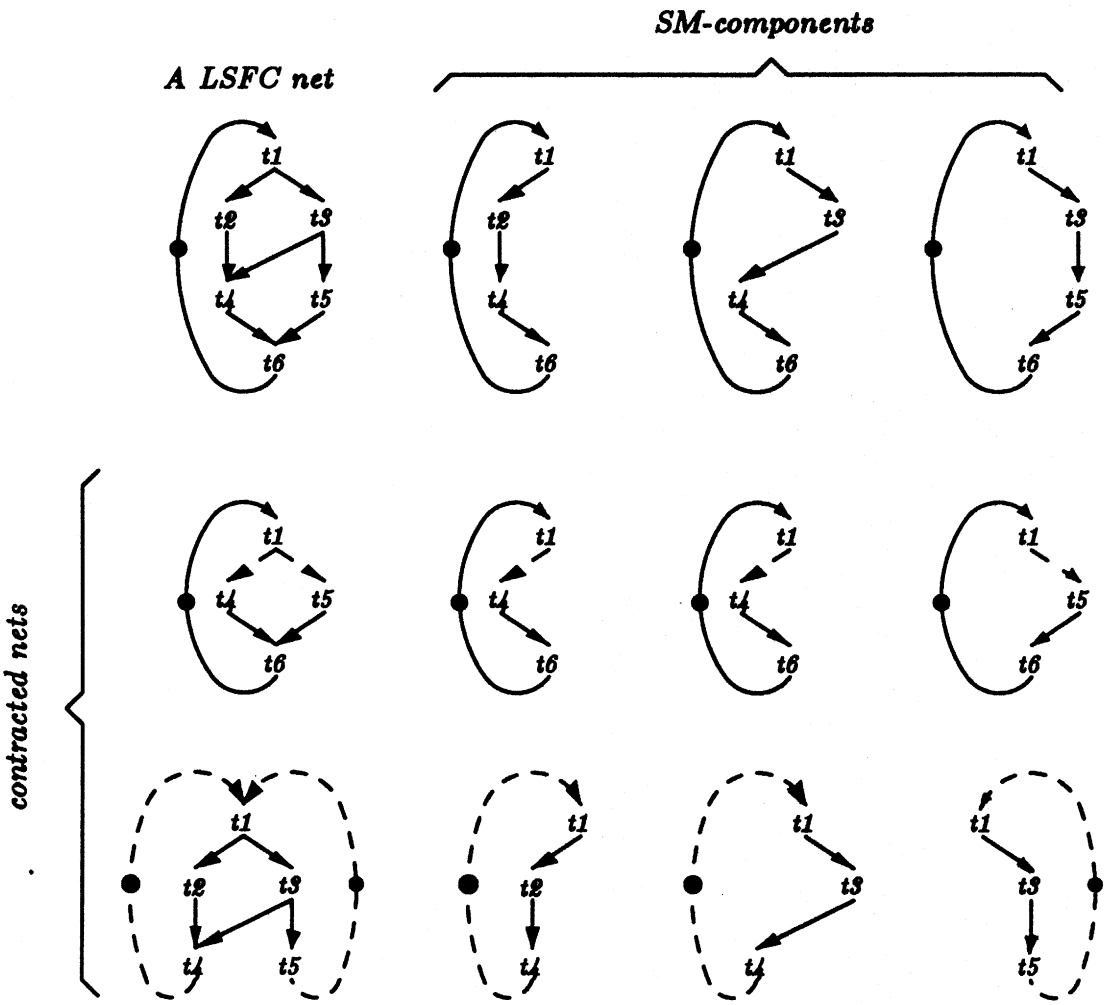


Figure 6.8: An example to illustrate the proof of Theorem 6.12. The first row contains a LSFC net Σ on the leftmost column, and three SM-components $\Sigma_1, \Sigma_2, \Sigma_3$. Below the first column are two contracted nets Σ_1', Σ_2'.

Proposition 6.13 *For every SM-component Σ_i as defined in the previous paragraph: $FS_i = FS'_{1,i} \parallel FS'_{2,i} \parallel \ldots \parallel FS'_{m,i}$, where $FS'_{j,i}$ denotes the set of firing sequences of the contracted SM-component $\Sigma'_{j,i} = \Sigma_i/T'_j$, as defined in the previous paragraph.*

By replacing the right-hand side of the above equation for each FS_i in the equation for FS, and using the fact that weaving is associative and commutative, we obtain the following expression for FS: (the terms in this expression have been rearranged such that each column i contains sets of firing sequences of the contracted nets of the SM-component Σ_i).

$$
\begin{array}{rcl}
FS & = & FS_1 \parallel \ldots \parallel FS_i \parallel \ldots \parallel FS_n \\
& = & \begin{array}{llllll}
FS'_{1,1} & \parallel \ldots \parallel & FS'_{1,i} & \parallel \ldots \parallel & FS'_{1,n} & \parallel \\
FS'_{2,1} & \parallel \ldots \parallel & FS'_{2,i} & \parallel \ldots \parallel & FS'_{2,n} & \parallel \\
\ldots & & \ldots & & \ldots & \parallel \\
FS'_{j,1} & \parallel \ldots \parallel & FS'_{j,i} & \parallel \ldots \parallel & FS'_{j,n} & \parallel \\
\ldots & & \ldots & & \ldots & \parallel \\
FS'_{m,1} & \parallel \ldots \parallel & FS'_{m,i} & \parallel \ldots \parallel & FS'_{m,n}. &
\end{array}
\end{array}
$$

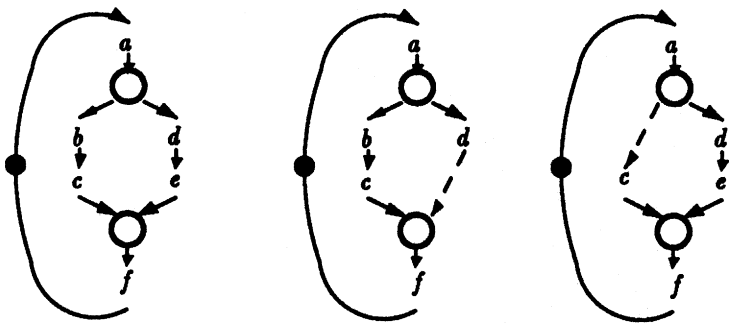
Notice that according to Proposition 6.13, each row j yields the set of firing sequences of the contracted net Σ'_j. Hence

$$FS = FS'_1 \parallel \ldots \parallel FS'_j \parallel \ldots \parallel FS'_m.$$

∎

Proof of Proposition 6.13. The proof is carried out for the case with two contracted nets: let Σ be a LS state machine and $\Sigma'_1 = \Sigma/T'_1$, $\Sigma'_2 = \Sigma/T'_2$ be its contracted nets such that the contractions are tr-preserving and $R'_1 \cup R'_2 \supseteq R$ (an example is given in Fig. 6.9). Let the sets of firing sequence of Σ, Σ'_1 and Σ'_2 be FS, FS'_1 and FS'_2, respectively. For convenience, let $FS = \{ \sigma_i \}$, $FS'_1 = \{ \gamma_i \}$ and $FS'_2 = \{ \beta_i \}$. Since the contractions are tr-preserving, according to Theorem 6.11, we have $FS'_1 = FS/T'_1$ and $FS'_2 = FS/T'_1$, implying that for every sequence $\sigma_i \in FS$, there exist one sequence $\gamma_i \in FS'_1$ and another $\beta_i \in FS'_2$ such that $\gamma_i = \sigma_i \lceil T'_1$, $\beta_i = \sigma_i \lceil T'_2$. (For example, in Fig. 6.9, for $\sigma_1 = abcf$, there exist $\gamma_1 = abcf$ and $\beta_1 = acf$; for $\sigma_2 = adef$, there exist $\gamma_2 = adf$ and $\beta_2 = adef$).

Since these nets are SMs, if sequence $t_1 t_2 \ldots t_n \in FS$ then $\langle t_i, t_{i+1} \rangle \in R$ for $1 \leq i < n$, and vice versa. Therefore if the causal relations R'_1, R'_2 cover R, then the above sequences



A LS state machine *Its contracted nets*

Figure 6.9: An example illustrating the weave of two contracted nets of a live-safe state machine.

γ_i and β_i "cover" γ_i, that is $\gamma_i \| \beta_i = \sigma_i$.[1]

Take any two firing sequences which do not correspond to part of the same simple cycle in Σ. Since Σ is a state machine, they represent choices in Σ. Then these two sequences correspond to part of two distinct cycles Ω_i and Ω_j which intersect, as indicated in Fig. 6.10. Let σ_i and σ_j denote their firing sequences, respectively. Let $X = (\sigma_i | \sigma_j)$, this is a set which can be constructed for any two sequences representing choices in Σ.

Since the contractions are tr-preserving, in Σ'_1 and Σ'_2 respectively, one can construct $X_1 = (\gamma_i | \gamma_j)$ and $X_2 = (\beta_i | \beta_j)$, where γ_i, β_i are defined in the first paragraph of the proof (see Fig 6.10). If we can show that $X = X_1 \| X_2$, then it follows that $FS = FS'_1 \| FS'_2$.

$$
\begin{aligned}
X_1 \| X_2 &= \{\, \sigma \in T^* \mid \sigma\lceil T'_1 \in \{\gamma_i, \gamma_j\} \ \wedge\ \sigma\lceil T'_2 \in \{\beta_i, \beta_j\} \,\} \\
&= \{\, \sigma \in T^* \mid (\sigma\lceil T'_1 = \gamma_i \vee \sigma\lceil T'_1 = \gamma_j) \wedge (\sigma\lceil T'_2 = \beta_i \vee \sigma\lceil T'_2 = \beta_j) \,\} \\
&= \{\, \sigma \in T^* \mid (\sigma\lceil T'_1 = \gamma_i \wedge \sigma\lceil T'_2 = \beta_i) \vee (\sigma\lceil T'_1 = \gamma_j \wedge \sigma\lceil T'_2 = \beta_j) \\
&\qquad \vee (\sigma\lceil T'_1 = \gamma_i \wedge \sigma\lceil T'_2 = \beta_j) \vee (\sigma\lceil T'_1 = \gamma_j \wedge \sigma\lceil T'_2 = \beta_i) \,\} \\
&= \{\, \sigma \in T^* \mid (\sigma\lceil T'_1 = \gamma_i \wedge \sigma\lceil T'_2 = \beta_i) \ \vee\ (\sigma\lceil T'_1 = \gamma_j \wedge \sigma\lceil T'_2 = \beta_j) \,\} \cup \\
&\quad \{\, \sigma \in T^* \mid (\sigma\lceil T'_1 = \gamma_i \wedge \sigma\lceil T'_2 = \beta_j) \ \vee\ (\sigma\lceil T'_1 = \gamma_j \wedge \sigma\lceil T'_2 = \beta_i) \,\} \\
&= \{\, \sigma_i, \sigma_j \,\} \cup \{\} = \{\, \sigma_i, \sigma_j \,\} = X.
\end{aligned}
$$

[1] It is more correct to write $\{\gamma_i\} \| \{\beta_i\} = \{\sigma_i\}$ instead; however the above simplified notations are used for the sake of clarity.

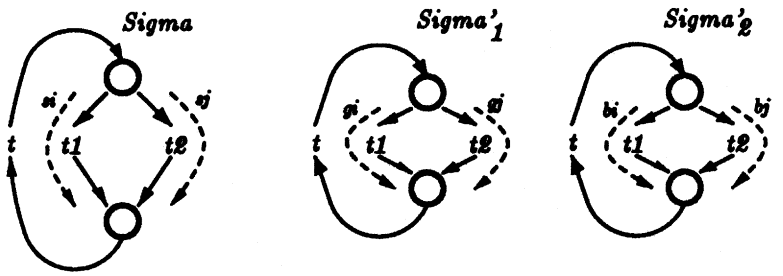


Figure 6.10: Proof of Proposition 6.13.

In the last line above, we need to show that

$$\{\ \sigma \in T^* \mid (\sigma\lceil T_1' = \gamma_i \wedge \sigma\lceil T_2' = \beta_j)\ \vee\ (\sigma\lceil T_1' = \gamma_j \wedge \sigma\lceil T_2' = \beta_i)\ \} = \{\}.$$

Let $Y = \{\ \sigma \in T^* \mid (\sigma\lceil T_1' = \gamma_i \wedge \sigma\lceil T_2' = \beta_j)\ \}$ and $Z = \{\ \sigma \in T^* \mid (\sigma\lceil T_1' = \gamma_j \wedge \sigma\lceil T_2' = \beta_i)\ \}$. Below we show that $Y = \{\}$; exactly the same technique can be used to show that $Z = \{\}$.

First, note that since Ω_1 and Ω_2 are two distinct intersecting cycles, there must exist some t which belongs to both cycles, some t_1 which belongs solely to Ω_1 and and some t_2 which belongs solely to Ω_2, as indicated in Fig. 6.10. Furthermore, the conditions that (i) contractions are tr-preserving ***and*** (ii) $R_1' \cup R_2'$ covers R together imply that there exist such t, t_1, t_2 which belong to both Σ_1' and Σ_2' (see Fig. 6.10): Condition (i) indicates that the structure of the net is preserved. Recall from Lemma 6.5 that it is illegal to remove all transitions on either branch of the contracted nets, therefore there must be some t_1, t_2 which remain in the left and right branches of the contracted nets. Condition (ii) implies that there must exist such t_1, t_2 which belong to both contracted nets: if no such t_1, t_2 exist, $R_1' \cup R_2'$ does not cover R. Therefore, we have (i) $\{t, t_1, t_2\} \subseteq T_1' \cap T_2'$ and (ii) sequences $\gamma_i = \ldots t \ldots t_1 \ldots$, $\beta_j = \ldots t \ldots t_2 \ldots$

From the definition of Y : $\sigma \in Y \ \Rightarrow\ \sigma\lceil T_1' = \gamma_i \wedge \sigma\lceil T_2' = \beta_j$. From (i) and (ii), it follows that

$$\sigma\lceil\{t, t_1, t_2\} = \ldots t t_1 \ldots \quad \text{and} \quad \sigma\lceil\{t, t_1, t_2\} = \ldots t t_2 \ldots .$$

Clearly both equalities cannot hold simultaneously and hence, no such sequence σ can exist, thus implying that $Y = \{\}$. ∎

6.4 Application to Signal Transition Graphs

Given a STG, one can derive from it a state graph by following the procedure outlined in a previous chapter. In order for the implementation obtained from the state graph to be deadlock-free and hazard-free, the STG specification is required to satisfy *liveness* and *persistency*. As discussed earlier, from the state graph, one can proceed to implementation by determining the *network function* which consists of a set of logic functions, one for each signal. The logic function of a signal can be obtained by determining, for every state in the state graph, the implied value of that signal; the logic function is precisely the set of all implied values of a signal.

There is a better alternative to the approach outlined above, being that of decomposition. In contrast to other decomposition methods such as those applied to the FSM model, a very simple alternative exists for state graphs derived from STGs. This alternate decomposition technique uses the *causal relation* in the STG to decompose its state graph, as described below.

Let Σ_J be a STG and Φ_J its state graph, both defined on a set of signals J which can be partitioned into $J = J_I \cup J_N \cup J_O$—the sets of input, internal and output signals, respectively. For every signal $i \in J$, the *input set* of i, denoted by $I(i)$, is defined as

$$\begin{aligned} I(i) &= \{j \in J \mid j_* R i_*\} \\ &= \{j \in J \mid j_+ R i_+ \vee j_+ R i_- \vee j_- R i_+ \vee j_- R i_-\} \end{aligned}$$

where R is the causal relation in Σ_J (In Chapter 5, we have provided the justification for relating this subset $j_* R i_*$ of the causal relation to the set of input of i). Thus $I(i)$ is the set of signals whose transitions *cause* transitions of signal i.

Since in Σ_J, $I(i)$ constitutes the set of signals whose transitions cause those of i, the *logic element* i can be implemented as a logic function with one output variable i and input variables in $I(i)$. Since input signals are given, we need be concerned only with the implementation of *non-input* signals, i.e. those in $J_{NI} = J_N \cup J_O$. Thus a straightforward decomposition algorithm for state graphs consist of the following steps.

Algorithm 6.14 (Decomposition of state graphs) *Let Σ_J be a STG and Φ_J be its state graphs. For every non-input signal $i \in J_{NI}$, let $J'(i) = \{i\} \cup I(i)$ and $T'(i) =$*

$J'(i) \times \{+,-\}$. *The set of decomposed state graphs of* Φ_J *is given by*

$$\{\ \Phi'(i) \mid i \in J_{NI}\ \},$$

where $\Phi'(i) = \Phi_J/T'(i)$, *the* $T'(i)$*-contracted state graph of* Φ_J.

It is easy to see that if that if the STG has a consistent state assignment, then so do the contracted state graphs obtained from the above algorithm. This is due to the fact that in a state graph, if a transition t is removed, then so is $\bar{t}$ as they are transitions of the same signal. Thus in every cycle in a state graph, every remaining pair of transitions $x, \bar{x}$ still alternate. Thus,

Lemma 6.15 *Let* Φ_J *be a state graph and* $\{\ \Phi'(i)\ \}$ *its set of contracted state graphs, as defined above. Then if* Φ_J *has a consistent state assignment, every* $\Phi'(i)$ *has a consistent state assignment.*

Based on the results developed in the previous section, there is a better way to determine the contracted state graphs. This is carried out by not performing contraction directly on the state graph Φ_J, but by first obtaining contracted *nets* from the STG Σ_J. These will produce the same contracted state graphs if contractions on nets preserve the temporal relation. Hence, the following decomposition algorithm based on net contraction is more efficient.

Algorithm 6.16 (Decomposition by net contraction) *Let* $\Sigma_J = \langle P, T, F, M_0 \rangle$ *and* $\Phi_J = \langle S, T, \delta, s_0 \rangle$ *be a STG and its state graph. For every non-input signal* $i \in J_{NI}$, *let*

$$J'(i) = \{i\} \cup I(i) \text{ and } T'(i) = J'(i) \times \{+,-\}.$$

Then the set of contracted state graphs $\{\ \Phi'(i) \mid i \in J_{NI}\ \}$ *can be obtained as follows. For* $i \in J_{NI}$,

(a) *Determine* $\Sigma'(i) = \Sigma_J/T'(i)$ *and its state graph* $\Phi(\Sigma'(i))$.

(b) *If* $T'(i)$*-contraction is tr-preserving then* $\Phi'(i)$ *is identically* $\Phi(\Sigma'(i))$. *Otherwise, add appropriate signals from* $j \in J - J'(i)$ *to* $J'(i)$ *such that the resulting contraction becomes tr-preserving. Then perform step (a) again.*

An important note. In a contracted net $\Sigma'(i)$ (defined over the subset of signals $J'(i) = \{i\} \cup I(i)$) we consider i as the *only* output signal, all signals in $I(i)$ are input. Hence in $\Sigma'(i)$, $I(i) \times \{+,-\}$ constitutes the set of transitions of input signals, while $\{i_+, i_-\}$ the set of transitions of output signals. This fact has important implication concerning persistency of contracted nets: in a contracted net for signal i, we only need to verify the persistency of transitions of i but not those of $I(i)$ as the latter are assumed to be persistent. Their persistency must be guaranteed by other contracted nets which represent other part of the control circuit. An example illustrating this point can be found in Chapter 7.

Chapter 7

A Design Example

As mentioned in Chapter 1, we advocate the use of STGs as a tool for specification and direct realization of distributed control modules which form the control structure of a system organized around the distributed control principle. We have been able to provide some real proofs of our approach by using STGs in the design of complete concurrent VLSI systems. These include a self-timed packet router with a maximum measured throughput rate of 22 Mbytes/sec [10] and a self-timed FIFO queue with a novel distributed organization and a measured throughput rate of 4 Mbytes/sec [9].

In this chapter, we stress another application area of asynchronous self-timed logic which is more conventional than the distributed organization proposed above. As an example, we examine the design of a self-timed controller for an A-to-D converter. First, this example serves to demonstrate that STGs can be useful for designing asynchronous logic in general, and secondly it is sufficiently complex to highlight most of the important and interesting features of our synthesis approach. This design was first published in [8].

Asynchronous control logic has found applications mostly in areas where the system inputs are inherently asynchronous. Some examples are vision VLSI systems [48] and interface circuits to asynchronous peripheral devices such as a disk drive. In these systems, asynchronous control circuits provide a modular interface which greatly facilitates system integration. In other cases, asynchronous circuits are almost indispensable and provides the highest operation rate possible; these include timing chains in dynamic and static memory devices [54] and even those used in synchronous microprocessors to generate extra cycles [22]. Given its usefulness, asynchronous logic has not been popular because of the

difficulties in its design and maintenance. In Chapter 1, several well-known problems with the Finite State Machine model and its implementation of asynchronous circuits have been summarized.

This chapter is organized as follows. In Section 1, we briefly compare different design alternatives for A/D converters and discuss the advantage of one with self-timed operation. The behavior of the self-timed controller for the A/D converter is then presented and the construction of a STG specification is described. In Section 2, the synthesis procedure and implementation of the controller are discussed in detail. Finally, Section 3 provides a few remarks on the use of STGs for direct synthesis of control circuits.

7.1 Specification of the Controller

A/D converters are subject to synchronizer failure because they make use of amplifiers as comparators; these are either high-gain or regenerative bistable amplifiers. When the input voltage v_{in} is close to the reference voltage v_{ref} (Fig. 7.1), the response time of the comparator becomes unbounded and its output may take an unbounded amount of time to settle at a valid voltage level (0 or 1). This type of failure has been observed in commercial A/D converters [50]. The synchronizer problem has been studied extensively, and it is well-known that if the circuit is required to produce a valid output within a certain time then there is a finite probability $\mathcal{P}$ that the output will be invalid at that time [13]. $\mathcal{P}$ decreases exponentially as the time allowed for the synchronizer to resolve is increased. For an N-bit converter using regenerative comparators, the analysis in [21] gives the following lower bounds on the worst-case conversion time T_{WC} in terms of the fault probability $\mathcal{P}$. For flash converters, $T_{WC} > N \ln 2 - \ln \mathcal{P}$, where T_{WC} is normalized to some time constant of the comparator. For clocked successive approximation converters, $T_{WC} > N(N \ln 2 - \ln \mathcal{P})$, simply because they take N steps to perform one conversion. For self-timed successive approximation converters, not all conversion steps are marginally close to the reference voltage, hence some conversions will be fast and some slow. As shown in [21], the self-timed successive approximation converter becomes significantly faster than clocked successive-approximation converters for very low $\mathcal{P}$ and large N.

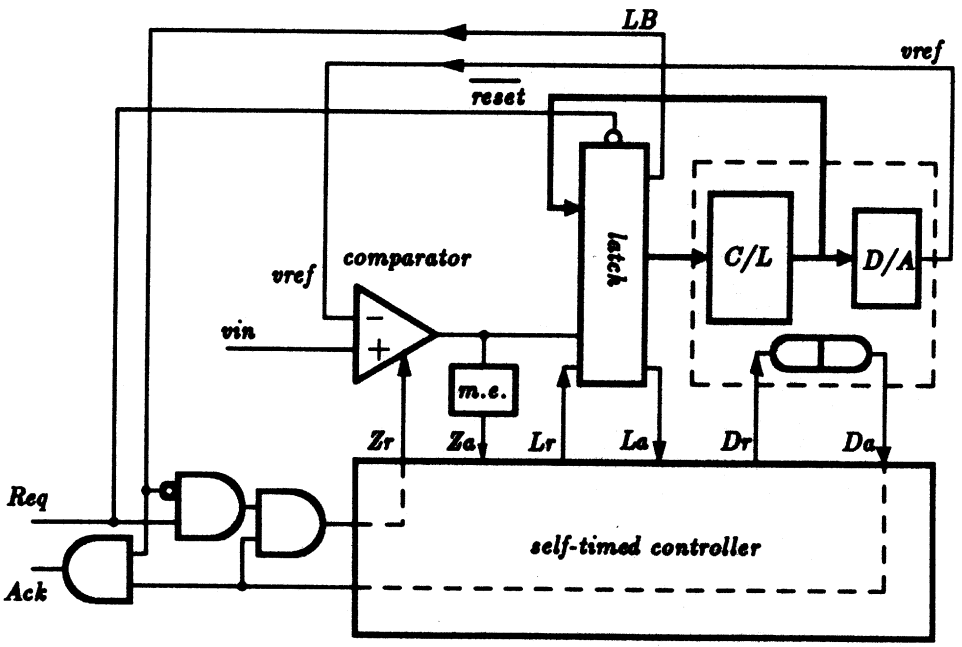


Figure 7.1: Block diagram of the successive-approximation A/D converter.

7.1.1 Behavior Specification

The block diagram of the successive approximation A/D converter is shown in Figure 7.1. The comparator senses the difference between the input voltage v_{in} and the reference voltage v_{ref} and produces a 1-bit result. The comparator has a control input Z_r which balances it at the beginning of each conversion when Z_r makes a 0-to-1 transition, and initiates a comparison when Z_r makes a 1-to-0 transition. It also has a *mutual-exclusion* (m.e.) circuit [32] whose output is active (=1) only when the comparator output is valid. This circuit is required because the comparison time is a function of the difference between the input voltage and the reference voltage; the smaller the difference, the longer the time it takes for the comparator to decide. This is the familiar phenomenon due to metastability [13].

The latch and the combinational logic form a finite state machine performing the successive approximation algorithm. Note that this machine operates in *pulse mode*, a mode of operation different from that of the self-timed controller we are synthesizing. Due to the fact that this machine performs many data-dependent operations, it is more economical and straight-forward to implement it in pulse mode instead of as a standard Huffman

asynchronous state machine. Data are latched on the rising transition of signal L_r and held in the latch after L_r goes low. Signal L_a goes high as soon as data are latched, and goes low shortly after L_r goes low. The reset input of the latch is controlled by signal $\overline{Req}$, so that when *Req* is low, outputs of the latch are reset to the appropriate initial values. Signal *LB* is the *Last-Bit* signal which goes high when the converter has determined the last bit of the digital word. The D/A converter at the right of the diagram accepts the digital word produced by the state machine and generates the analog voltage v_{ref}. The combined delay of the combinational logic and the D/A converter is matched by some delay circuit from D_r to D_a. While it is possible to accomplish this timing constraint in a speed-independent manner using dual rail coding [32], a simple delay circuit is more justifiable from an engineering standpoint.

Initially, the state of the system is

$$Req = Ack = Z_r = Z_a = L_r = L_a = 0, D_r = D_a = 1.$$

Since $Req = 0$, the latch is initialized with $LB = 0$. Thus, the and-gate whose input is *Req* is enabled and the and-gate whose output is *Ack* is disabled. When *Req* is raised, Z_r will go high and initiate a cycle of the successive-approximation algorithm. After each cycle, D_a will restart another cycle by causing Z_r to go high again. This is repeated until *LB* becomes high during the last cycle. This will cause *Ack* to be raised instead of Z_r when D_a goes high. After that, *Req* drops in response to *Ack*, resetting *LB* and in turns *Ack* to low. At this point the circuit returns to its initial configuration for the next conversion.

7.1.2 STG specifications

A Signal Transition Graph describing the operation of the self-timed controller is shown in Fig. 7.2. Since the circuit operation is totally deterministic, i.e. there is no data-dependent operation, no places are drawn explicitly. The arcs represent the causal relation R discussed earlier. In this chapter, we will use aRb and $a \rightarrow b$ interchangeably in order to improve readability. Intuitively, aRb (*a causes b*) can be understood as as a *timing* or *sequencing constraint* between occurrences of two signal transitions.

The two bold arcs in Fig. 7.2 are not part of the sequencing requirement of the circuit and they can be ignored for the moment; they are *persistency constraints* added to the

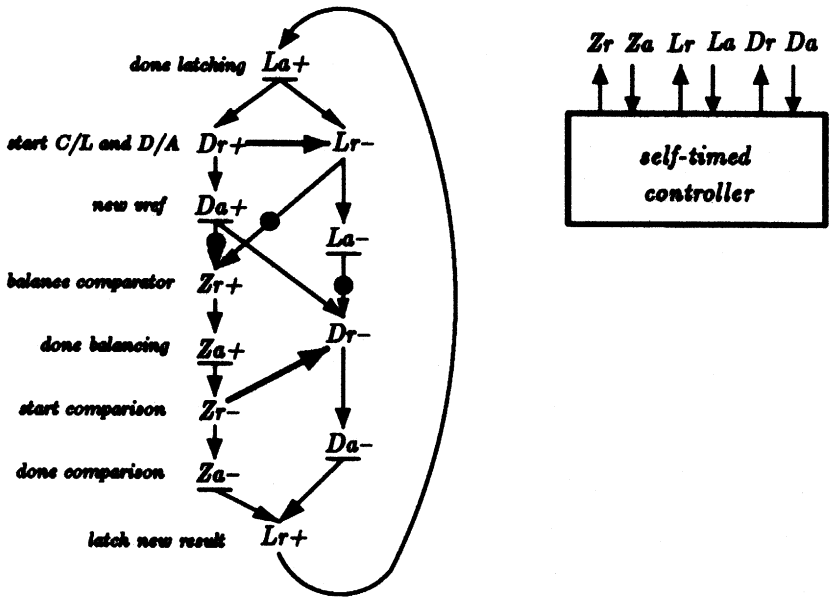


Figure 7.2: A STG specification of the self-timed controller.

STG to ensure persistency, as discussed in Chapters 2 and 5. There are two subtle timing constraints required for the correct coordination of data and control signals: The constraint $L_{r-} \rightarrow Z_{r+}$ guarantees that the gating signal of the latch is turned off before the comparator changes its output value, so that there is no possibility of latching the comparator output while it is changing. The constraint $D_{a+} \rightarrow Z_{r+}$ ensures that a comparison is initiated only after a new value of v_{ref} is available (as signified by D_a going high).[1]

Finally, transitions D_{r-}, D_{a-}, L_{r-} and L_{a-} in Fig. 7.2 are simply reset transitions of the *reset signaling* handshake protocol [32]. For the control circuit which we specify, event occurrences are signaled over control links, using the reset signaling protocol. Usually, an occurrence of an event is signaled by a positive transition on the ready wire of the control link; its acknowledgment is signaled by a positive transition on the acknowledge wire of the control link. The signals on these links are then reset through negative transitions before the occurrence of the next event can be signaled. In this communication discipline, a transition on the *acknowledge* wire can only occur in response to a transition on the *ready* wire and vice versa. As discussed in Chapter 4, for an input link $\{I_r, I_a\}$ where I_r is an input *ready* and I_a an output *acknowledge*, this communication interface to the external

[1] A specification with more concurrency can be obtained by requiring only $D_{a+} \rightarrow Z_{r-}$, thus allowing the comparator to be balanced while a new v_{ref} is evaluated. However, we will use this specification with less concurrency to illustrate the design procedure.

world is specified in a STG by the pair of constraints $\{I_{a-} \rightarrow I_{r+}, I_{a+} \rightarrow I_{r-}\}$. Similarly for an output link $\{O_r, O_a\}$ where O_r is an output *ready* and O_a an input *acknowledge*, its corresponding set of constraints is $\{O_{r+} \rightarrow O_{a+}, O_{r-} \rightarrow O_{a-}\}$. These interface constraints are an example of our design rules described in Section 4.3 which states that in a STG, ***every transition of an input signal has exactly one transition which directly precedes it, and this transition must be that of an output signal.*** Transitions of input signals to the circuit are underlined to distinguish them from those of non-input signals. This reflects the fact that transitions of input signals are generated externally, whereas non-input ones are generated internally by the circuit.

7.2 Synthesis from STG Specification

The STG specification of the controller in Fig. 7.2 is a concise description of its operation based on the causal relation between signal transitions in the circuit. In order to obtain a logic implementation of the circuit from this specification, it must be ensured that the STG specification satisfies liveness and persistency. Liveness of the STG implies that its state graph is strongly connected and has a consistent state assignment, hence the circuit realization is free from deadlock. Persistency implies that the circuit realization is hazard-free. To summarize, the synthesis procedure from a STG specification consists of the following steps:

(a) Meeting liveness and persistency. Checking for the state assignment problem and introducing internal signals if required.

(b) Decomposing the STG into contracted nets and obtaining their state graphs.

(c) Determining the logic equation for every non-input signal from its state graph.

7.2.1 Meeting liveness and persistency

It is easy to verify that the STG specification of the control circuit (Fig. 7.2) satisfies the liveness conditions: it is strongly connected and for every transition t, there exists a simple cycle containing both t and $\bar{t}$. The second condition also indicates that the state graph has a consistent state assignment.

One can view the construction of a STG from a control circuit's behavior as a specification of the presence or absence of sequencing constraints between control events. Such a preliminary specification will usually contain non-persistent transitions due to interactions between concurrent signal transitions, as discussed in Chapter 5. The addition of persistency constraints essentially serve to limit the allowed transition sequences to a subset of the original set such that none can produce non-persistent behavior. Sometimes the original specification may need to be altered slightly for persistency to be satisfied.

Without the bold arcs, in the above STG, transitions D_{r+} and Z_{r+} are non-persistent since the following conditions are true (Theorem 5.5):

$$(a)\ L_{a+} \rightarrow D_{r+} \textit{ and } \{L_{a-}, D_{r+}\} \in \mathbf{co},$$
$$(b)\ D_{a+} \rightarrow Z_{r+} \textit{ and } \{D_{a-}, Z_{r+}\} \in \mathbf{co}.$$

To eliminate non-persistency in case (a), we have to add the persistency constraint $D_{r+} \longrightarrow_p L_{a-}$. This condition can be satisfied by introducing the bold arc $D_{r+} \rightarrow L_{r-}$. In this case, we cannot use the arc $D_{r+} \rightarrow L_{a-}$ because L_a is an input signal to the control circuit, and as mentioned earlier, its transition is allowed to have exactly one predecessor.

In case (b), arc $Z_{r+} \rightarrow D_{r-}$ could be used to satisfy the persistency constraint $Z_{r+} \longrightarrow_p D_{a-}$ which will ensure the persistency of Z_{r+}. However, this arc would necessitate the addition of another arc $D_{r-} \rightarrow Z_{r-}$ to guarantee the persistency of D_{r-}. Instead of using this pair of arcs, we chose the bold arc $Z_{r-} \rightarrow D_{r-}$ which also satisfies the above persistency constraint. This example shows that there is more than one choice for satisfying the persistency constraints; hence, may help the designer make the best choice.

With the addition of these new arcs, some of the sequencing constraints become redundant. For instance, the constraint $L_{a+} \rightarrow L_{r-}$ is already satisfied by the existing pair $L_{a+} \rightarrow D_{r+}$ and $D_{r+} \rightarrow L_{r-}$. One can thus modify the specification by removing these redundant constraints. Two redundant constraints $L_{a+} \rightarrow L_{r-}$ and $D_{a+} \rightarrow D_{r-}$ can be removed from Fig. 7.2 to produce the STG of Fig. 7.3a.

At this point, we have modified the original specification to produce a STG which satisfies liveness and persistency as part of the sequencing requirement. We also need to check for the possibility of non-persistency due to state assignments. In Fig. 7.3a, one can immediately detect an R-path $(Z_{r+}, Z_{a+}, Z_{r-}, Z_{a-})$ which results in a state-assignment which exhibits non-persistency. Intuitively, in the absence of any other intervening transition, consecutive rising and falling transitions of Z_a and Z_r take the circuit back to the

same state. More specifically, there are two p-cuts

$$C_p = \{\langle D_{a+}, Z_{r+}\rangle, \langle L_{r-}, Z_{r+}\rangle, \langle L_{r-}, L_{a-}\rangle\},$$
$$C'_p = \{\langle Z_{a-}, L_{r+}\rangle, \langle Z_{r-}, D_{r-}\rangle, \langle L_{r-}, L_{a-}\rangle\}$$

as indicated by the dashed arcs in Fig. 7.3a such that $[C_p, C'_p]$ forms a complementary set. Since Z_{r+} satisfies the conditions of Lemma 5.8:

$$Z_{r+} \in (C_p \cdot \div C'_p \cdot) : (\cdot Z_{r+}) \subseteq C_p \ \wedge \ Z_{r+} \notin T_I,$$

Z_{r+} is non-persistent. This is illustrated in Fig. 7.3b, the binary representations of both M and M' are 000111, where M and M' are markings corresponding to p-cuts C_p and C'_p, respectively. Starting from M, the firing sequence $Z_{r+}Z_{a+}Z_{r-}Z_{a-}$ leads to marking M' at which, in addition to L_{a-} being enabled, Z_{r+} is also enabled. This gives rise to non-persistency of Z_{r+}. On the other hand, note that even though L_{a-} satisfies the conditions of Lemma 5.7, it can be assumed to be persistent because it is a transition of an input signal.

As mentioned earlier, in order to eliminate non-persistency due to state assignment, one can introduce an additional *internal* signal to permit the distinction between the binary representations of markings M and M'. Thus, one could introduce a signal x, and insert a transition of x, e.g. x_+, into the middle of the R-path $(Z_{r+}, Z_{a+}, Z_{r-}, Z_{a-})$ to obtain $(Z_{r+}, Z_{a+}, x_+, Z_{r-}, Z_{a-})$. This in effect removes all complementary sets formed by distinct p-cuts from the STG. In order to ensure liveness of the modified STG, one has to determine a place to insert x_- as well. However, to carry this task out effectively, we need to consider the next synthesis step of decomposition.

As outlined above, we can decompose the STG into a number of contracted nets and then determine their state graphs. Clearly, one could take the alternative approach of deriving the state graph directly from the STG (with the addition of signal x) and determining logic equations from it. However, this would involve a total of seven signals, and every state will be a binary representation of the set $\langle Z_r, Z_a, L_r, L_a, D_r, D_a, x\rangle$. Without decomposition, at worst, the logic equation of any non-input signal Z_r, L_r, D_r or x may depend on all seven variables. Obviously, this is grossly inefficient. The alternative approach of decomposition can be carried out in a straightforward manner by using the causal relation R to produce an efficient implementation, as discussed below.

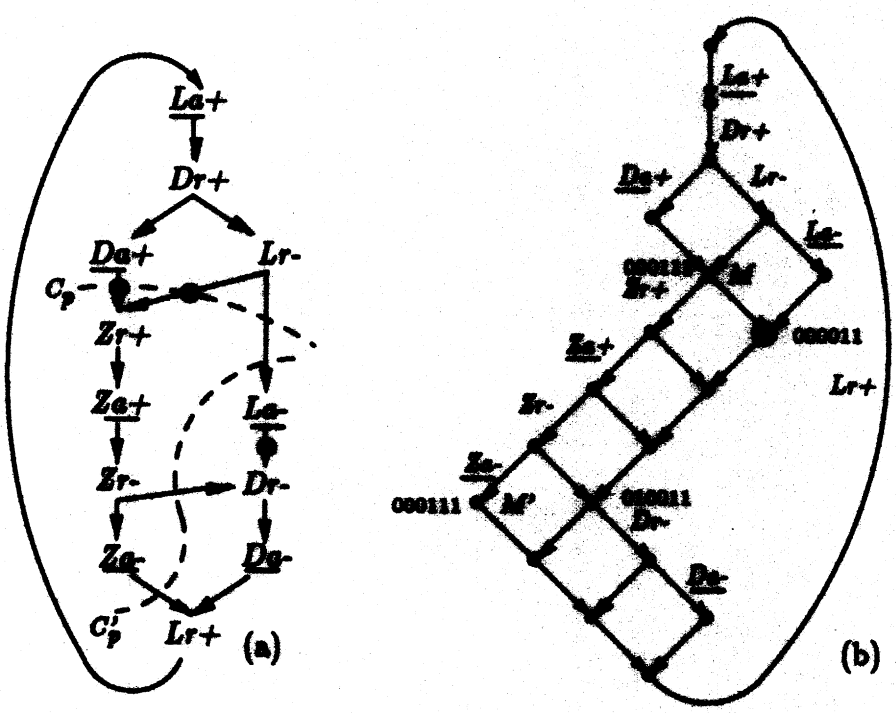


Figure 7.3: (a) The original STG specification with persistency constraints added. (b) Its state graph indicates that there exist markings M and M' with identical state assignments, which cause Z_{r+} to be non-persistent.

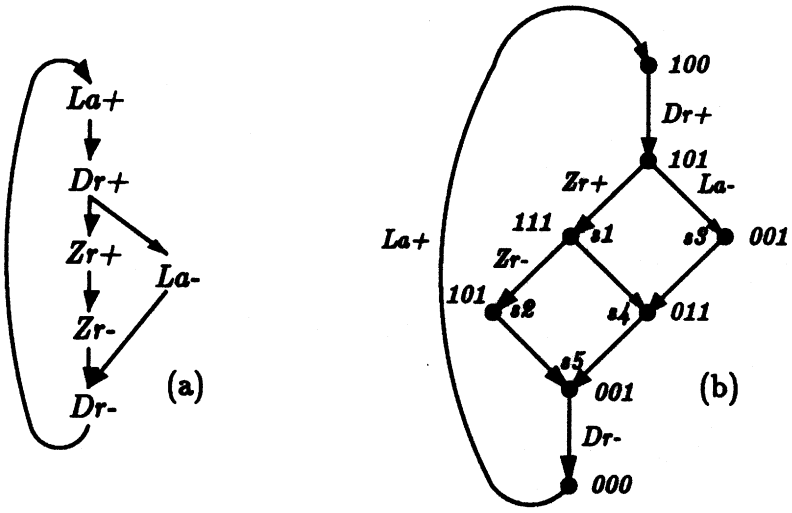


Figure 7.4: (a) The contracted net of D_r derived from the STG in Fig. 6.3a. (b) State graph of the contracted net in (a).

7.2.2 Implementation using decomposition

We apply the synthesis procedure as outlined earlier to produce a circuit, starting from the STG in Fig. 7.3a. It will be demonstrated that when this STG is decomposed into reduced graphs, the state graph of one of them is non-persistent due to the lack of state information. As suggested earlier, this requires the addition of an internal signal x to the STG specification. In the following presentation, the synthesis procedure will consist of two passes, with the latter producing the final circuit implementation.

First Pass. Fig. 7.3a shows that L_a, D_a, Z_a are input signals to the circuit, while the rest are non-input signals whose logic equations are to be determined. The STG in this figure has three non-input signals D_r, L_r, Z_r, and their input sets are $I(D_r) = \{L_a, Z_r\}$, $I(L_r) = \{D_r, D_a, Z_a\}$ and $I(Z_r) = \{L_r, D_a, Z_a\}$.

The contracted net of D_r (denoted by $\Sigma'(D_r)$) and its state graph are shown in Fig. 7.4; each vertex in the state graph is a binary vector representing signals in the set $\langle L_a, Z_r, D_r \rangle$. From state $s_3 = 001$, the consecutive occurrences of Z_{r+} and Z_{r-} take the circuit to state 011 and then back to 001. (A similar case occurs starting from state 101.) Correspondingly, in the contracted net of D_r, there is an R-path of consecutive transitions of signal Z_r which

directly precedes the output transition D_r: (Z_{r+}, Z_{r-}, D_{r-}). According to this state graph, logic element D_r is implemented such that whenever it is in state s_5, it will cause transition D_{r-} to occur. However, the state assignment results in $s_3 = s_5$. Hence, whenever the circuit is in either state, both transitions Z_{r+} and D_{r-} are enabled. In which case, the transition sequence $D_{r+}L_{a-}D_{r-}$ may take place instead of the correct sequence $D_{r+}L_{a-}Z_{r+}Z_{r-}D_{r-}$, and the circuit malfunctions. Moreover, transition D_{r-} is non-persistent because while it is enabled in state s_3, the occurrence of Z_{r+} brings the circuit to state s_4 in which D_{r-} is no longer enabled.

The problem in this situation arises from the fact that it is impossible for the circuit to distinguish that s_3 and s_5 are supposedly different. In order to modify them into distinguishable binary states, we add another signal called x into the circuit. Since this problem shows up in the contracted net of D_r as a pair of consecutive transitions of the same signal Z_r, a transition of x, say x_+, is inserted between them. This requires that x_- also be added to preserve liveness. The contracted net D_r indicates that x_- must not be inserted (i) between the pair (D_{r+}, Z_{r+}) or (Z_{r-}, D_{r-}) because this only produces the same problem but with *two* pairs of consecutive transitions of the same signals, (ii) into the R-path (D_{r+}, L_{a-}, D_{r-}) because this makes x_+ and x_- concurrent and thus violates both liveness and persistency. Hence x_- must be inserted into the R-path (D_{r-}, L_{a+}, D_{r+}). Considering the STG of Fig. 7.3a, this means that x_- must be inserted into the path $(D_{r-}, D_{a-}, L_{r+}, L_{a+}, D_{r+})$. Furthermore, since transitions of input signals D_a, L_a and Z_a can have only one incident arcs coming from transitions of their corresponding *request* signals D_r, L_r and Z_r, x_- cannot be inserted in front of these transitions. Thus x_- can be inserted between (D_{a-}, L_{r+}) as shown in Fig. 7.5, or between (L_{a+}, D_{r+}). In this latest specification, x_+ does not directly precede transitions of signal D_r; however it must be used as an input to logic element D_r to eliminate hazards at signal D_r.

Finally, note that transition x_- can also be inserted between (L_{a+}, D_{r+}) in the STG in Fig. 7.3a. This would result in another STG specification which yields a slightly different implementation of the circuit. This fact indicates that the implementation is sensitive to the particular form of the STG, which is understandable because the state graphs extracted from STGs are unique state-based representations of the behavior of a circuit.

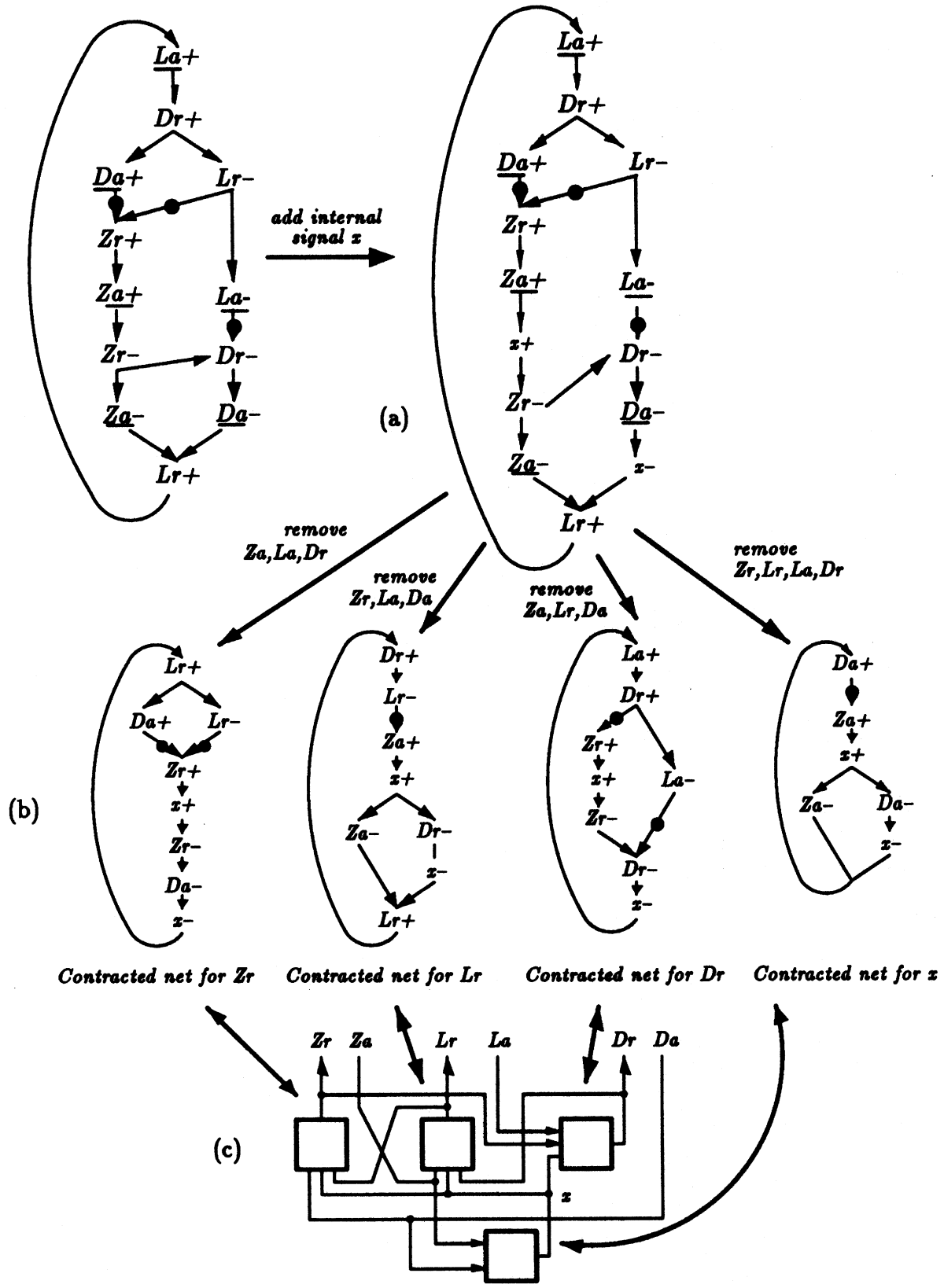


Figure 7.5: (a) The final STG with the addition of signal x. (b) Contracted nets for non-input signals Z_r, L_r, D_r and x. (c) Structure of the corresponding circuit.

Second Pass. The STG in Fig. 7.5a shows that there are four non-input signals D_r, L_r, Z_r and x, with input sets

$$\begin{cases} I(L_r)=\{D_r, Z_a, x\} \\ I(Z_r)=\{L_r, D_a, x\} \\ I(x) \ =\{Z_a, D_a\} \\ I(D_r)=\{L_a, Z_r, x\}. \end{cases}$$

The input set $I(D_r)$ contains signal x due to the reason just described. The contracted nets of D_r, Z_r, L_r and x are shown in Fig. 7.5b. At this point, we obtain the structure of the circuit using the input/output information of the constituent logic elements, as shown in Fig. 7.5c.

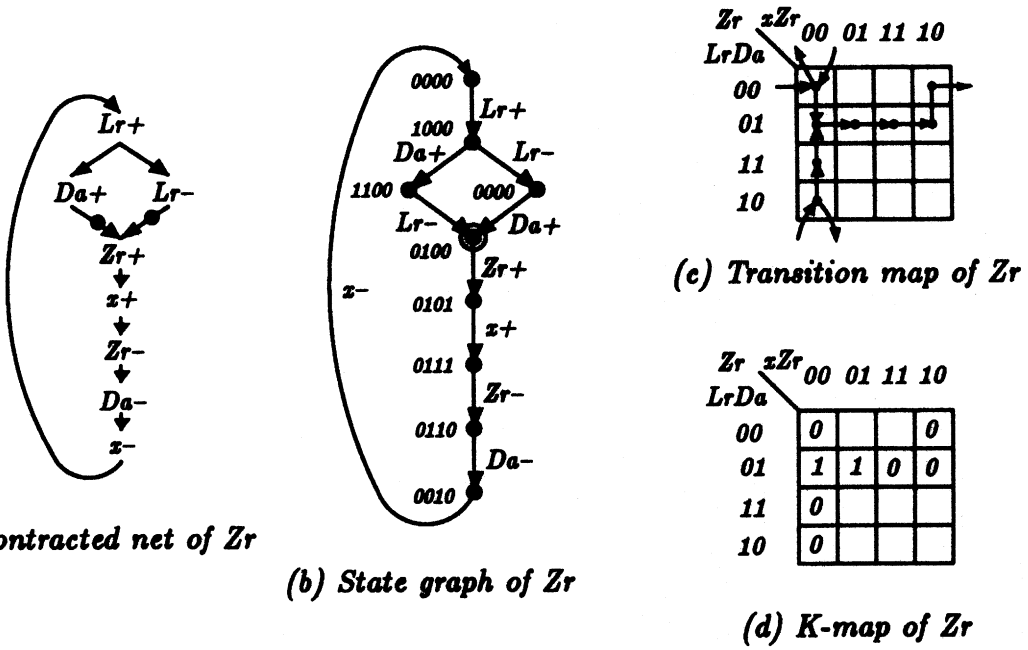
The final step in the synthesis process is to derive logic equations from contracted nets. This step is illustrated for signals Z_r and x. From the contracted net of Z_r (reproduced in Fig. 7.6a), one can derive its state graph (Fig. 7.6b) with states representing signals in $\langle L_r, D_a, x, Z_r \rangle$.

The state graph can be transferred to a type of K-map called *transition map*; the transition map of signal Z_r is shown in Fig. 7.6c. Each entry in this map corresponds to a state, which is a binary representation of the signals $\langle L_r, D_a, x, Z_r \rangle$; arcs between entries are simply transitions between states as given by the state graph. A K-map for Z_r can be obtained by replacing each entry (corresponding to a state) in the transition map with its implied value for Z_r, as discussion in Chapter 4 (Fig. 7.6d). For example, in state 0111, the implied value of Z_r is 0, thus this entry in the transition map is replaced by a 0. The logic equation of Z_r can be found from this K-map to be

$$Z_r = \bar{L}_r D_a \bar{x}.$$

Lastly, note that in the contracted net of Z_r, there is an R-path (L_{r+}, L_{r-}). However, in contrast to the previous case, it does not cause non-persistency. Its state graph (Fig. 7.6b shows that from state 0000, the firing sequence $L_{r+}L_{r-}$ leads back to state 0000. Thus both L_{r+} and D_{a+} are enabled in state 0000; however, since they are considered as *inputs* to logic element Z_r, they can be assumed to be persistent. The persistency of L_{r+} and D_{a+} has to be guaranteed by logic elements L_r and D_a, respectively.

In a state graph for an output signal i, it is often the case that a certain state s has more than one next-state. Hence we need to choose one for its implied value. Suppose



(a) Contracted net of Zr

(b) State graph of Zr

(c) Transition map of Zr

(d) K-map of Zr

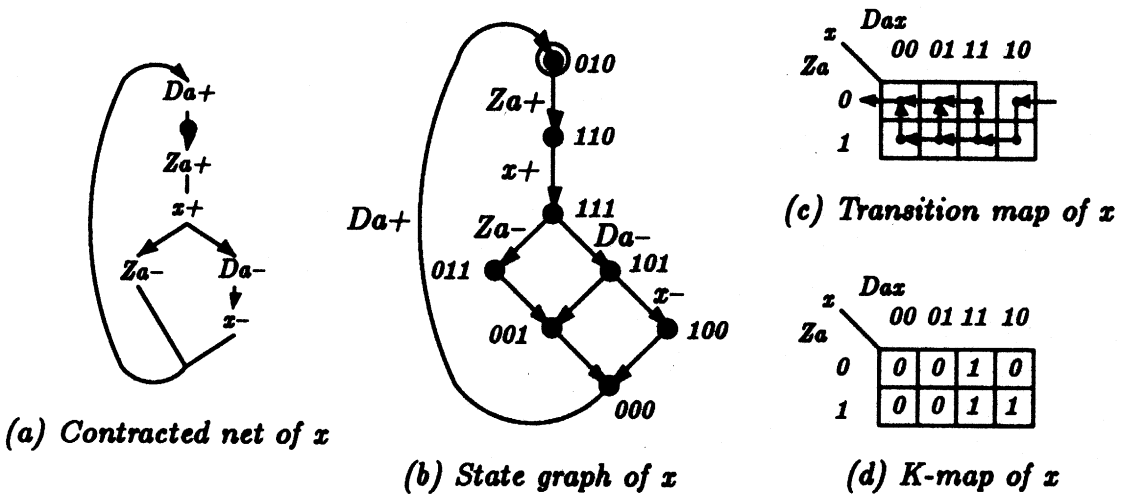
Figure 7.6: Steps in the transformation from a contracted net to the logic equation for signal Z_r.

that transitions t_1 and t_2 are enabled in s, leading to next-states s_1 and s_2, respectively. As described in Chapter 2, there are two cases:

- If neither t_1, t_2 is a transition of signal i ($t_1, t_2 \neq i_*$), then $s_1(i) = s_2(i)$. In this case, the implied value of s is unique. K-map entry corresponding to s is entered with this unique value $s_1(i)$.
- If either t_1 or t_2 is a transition of i, for instance, $t_1 = i_*$ and $t_2 \neq i_*$, then $s_1(i) \neq s_2(i) = s(i)$[2]. In this case, the next-state value of s is not unique. However, we require that the K-map entry corresponding to s be entered with $s_1(i)$, the implied value which results from the transition of signal i itself. If furthermore, t_1 is also enabled in state s_2, then t_1 is persistent and no hazard results. Otherwise, t_1 is non-persistent and the circuit has hazards. Note, however, that this will not happen if the STG is persistent.

In the state graph of x, state 101 has two next-states due to the concurrent transitions

[2]Note that the case with $t_1 = \bar{t}_2 = i_*$ is not possible as either t_1 or t_2 will not be consistent for state s.



(a) Contracted net of x

(b) State graph of x

(c) Transition map of x

(d) K-map of x

Figure 7.7: Steps in the transformation from a contracted net to the logic equation for signal x.

of x_- and Z_{a-} (Fig. 7.7a). In state 101, the implied value for x is chosen to be 0, as it results from the transition x_-. Fig. 7.7a shows that regardless of whether logic element x is in state 101 or 001, transition x_- will always occur next and the circuit behaves exactly the same. The transition map is shown in Fig. 7.7b. The K-map derived from this state graph is shown in Fig. 7.7c where the logic equation is found to be

$$x = Z_a D_a + x D_a.$$

This equation has the general form $x = S + x\overline{R}$ with $S = Z_a D_a$ and $R = \overline{D}_a$. Its implementation is a set-reset flipflop whose output is x, the set and reset inputs are $Z_a D_a$ and $\overline{D}_a$, respectively. In order for this implementation to work properly, it is required that $S.R = 0$ at all times.

Similarly, the same procedure can be applied to other contracted nets to obtain the logic equations for L_r and D_r. They are

$$\begin{aligned} L_r &= \overline{D}_r \overline{x} \overline{Z}_a \\ D_r &= Z_r + L_a + D_r \overline{x}. \end{aligned}$$

The equation for D_r can be rewritten as $D_r = S + D_r\overline{R}$ with $S = Z_r + L_a$ and $R = x$, and it is implemented as a set-reset flipflop. The contracted net of D_r in Fig. 7.5b shows that there is a time period during which both Z_r and x are high, causing both the *set* and

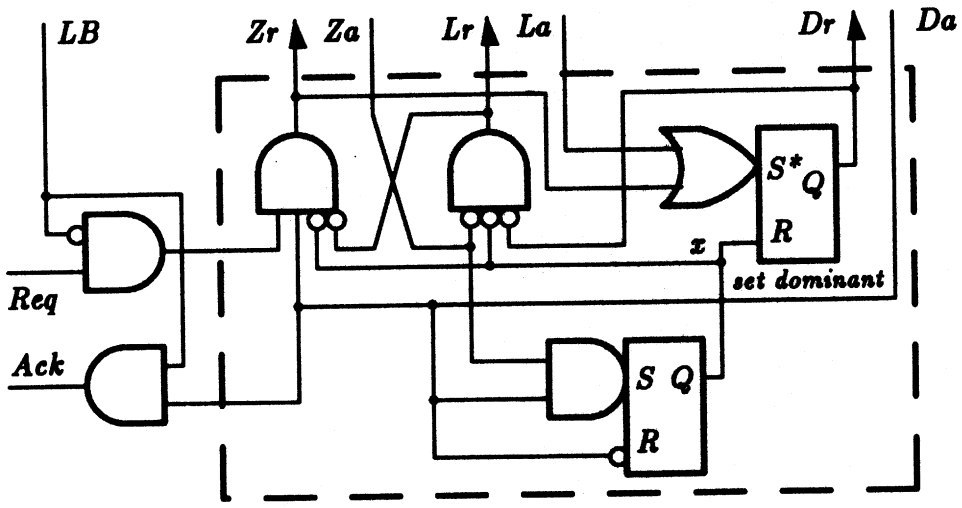


Figure 7.8: The final circuit realization of the self-timed controller. The controller's initial state is $Z_r = Z_a = L_r = L_a = x = LB = 0$, $D_r = D_a = 1$.

reset inputs of the D_r flipflop to be active. This violates the condition that $S.R = 0$ at all times. However, it also indicates that output D_r is not to be reset until after both Z_r and L_a go low, and therefore, until after the set input goes low. Therefore, this circuit can be implemented as a ***set-dominant*** flipflop (indicated by S^* in Fig. 7.8). One can also choose to implement D_r directly from the equation given above instead of a set-reset flipflop and not to worry about this particular detail.

Finally, by putting all these elements together, one obtains the control circuit for the A/D converter as shown in the dashed box in Fig. 7.8. The self-timed control circuit shown is speed-independent, i.e., it operates correctly with any combination of delays of logic gates, assuming that the internal feedback delays of the flipflops are negligible compared to other loop delays in the control circuit.

7.3 Summary

In this chapter, STGs have been used as a specification tool for asynchronous control circuits. A STG specification can be viewed as an interpreted Petri net in which each transition is identified with a signal transition in a hardware circuit. In the synthesis approach proposed, state graphs are generated from a STG and then used to derive logic

equations and hardware structures for the signals. In the above specification and design example, it has been shown how introducing additional constraints in a STG allows us to use level-sensitive hardware circuits instead of transition-sensitive hardware circuits in its implementation. These are precisely the persistency constraints which guarantee speed-independent (and therefore hazard-free) implementation.

A STG specification can thus also be viewed as a concise yet more abstract notation for specifying a class of state graphs. As specifically illustrated through examples above, most of the complex interactions of digital control circuits at the signal level can be lifted to our abstract representation using STG notations. At this higher level, one can guarantee live and hazard-free operation of control circuits by simply satisfying syntactic conditions on STGs. This, perhaps, is the most important point of our graph-based approach.

The module descriptions used in this chapter require only constructs for specifying sequencing and concurrency. There are other behaviors which exhibit conflict and data-dependent signal flow that would require additional STG constructs for their specification. The formulation and application of these latter constructs are presented in the next chapter.

Chapter 8

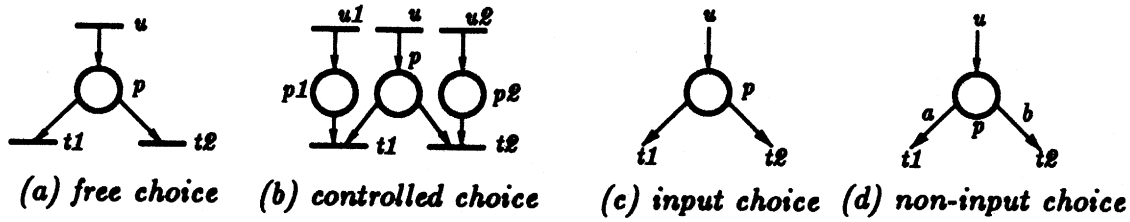
Signal Transition Graphs with Non-input Choices

8.1 Introduction

The Signal Transition Graphs considered so far belong to a class of interpreted *free-choice* nets. By restricting transitions which are in direct-conflict to those of input signals, *free-choices* can be used to specify (nondeterministic) *input choices* to a control module. Thus, in addition to sequential and concurrent operations, these STGs can also specify input choices. In practice, however, this ability is only of limited use. More often, one also needs to specify control operations involving choices of *internal* events—the particular choice of which control event to execute depends on the state of certain control variables. In this chapter, instead of purely free choices, we consider cases in which choices are controlled by the state of certain conditions (which hold due to the firing of some signal transition in the circuit). This gives rise to the class of nets with *controlled-choices*.

There are two alternatives for extending nets to permit the specification of controlled-choices, as illustrated in Fig. 8.1. We may choose a *structural* extension by using nets which are structurally more complex than free-choice ones, or we may choose a *behavioral* extension by developing new notations which permit the use of additional conditions to further restrict the sets of firing sequences of nets.

Fig. 8.1a is a free-choice: whenever p is marked with one token, both t_1 and t_2 are enabled and one is nondeterministically chosen to fire. Fig. 8.1c is the corresponding



(a) free choice *(b) controlled choice* *(c) input choice* *(d) non-input choice*

Figure 8.1: Structural and behavioral extension to the FC net model.

situation for STGs called an input choice, in which t_1 and t_2 are interpreted as transitions of input signals; the decision of which transition to fire is made externally to the system and hence appears to be nondeterministic.

A structural extension to free-choice nets to allow the specification of controlled-choices is depicted in Fig. 8.1b. This net specifies a controlled-choice if whenever transition u is about to fire (thus marking place p), a token must have already been present at either place p_1 or p_2. Then exactly one of t_1, t_2 is enabled and the enabled transition will fire—nondeterminism never arises in this situation. Fig. 8.1d illustrates the corresponding situation in STGs, called *non-input choices*, i.e., choices involving transitions of internal or output signals. In contrast to the input choice in Fig 8.1c where t_1 and t_2 are transitions of input signals, here they must be transitions of non-input signals. The arc labels a and b are *logical* variables which indicate, respectively, whether places p_1 and p_2 are marked at the moment p receives a token. For example, we can pick $a = \langle i, 1\rangle$, $b = \langle i, 0\rangle$ and $u_1 = i_+$, $u_2 = i_-$. Then a being true means that signal i becomes 1 after transition i_+ has fired; a being false means that signal i ceases to be 1 (due to the firing of i_-), etc.

As will be discussed in more detail, the behaviorally extended net in Fig. 8.1d can serve as a correct specification of non-input choices only if the following two conditions are met:

(i) Only one of the logical variables a and b can become true whenever p is marked. (If both can hold simultaneously when p is marked, this controlled-choice becomes a free-choice. Thus, free-choice nets can be considered as a special case of controlled-choice nets.)

(ii) Transitions whose firings cause a and b to hold are *not* concurrent with u (in the above example, these transitions are u_1 and u_2, respectively). This condition means that whenever p is marked, it is guaranteed that at least one of its output transitions,

t_1 or t_2, is enabled. Otherwise, if u_1, u_2 are allowed to fire concurrently with u, it is generally impossible to determine the state of variables a and b when p is marked. This condition is a fundamental requirement of the proposed behavioral extension; further discussion will be given when we present the unfolding algorithm in Chapter 9.

Under these provisions, controlled choices can be viewed as ***additional restrictions on the set of firing sequences*** specified by a FC net. As such, they are a behavioral extension to FC nets. The reason for avoiding a structural extension is that to date, an acceptable structure theory is available only for the class of free-choice nets. Techniques for structural analysis of more complex nets which are useful for our purposes have not been fully developed.

This chapter describes this extended class of interpreted nets called ***STGs with non-input choices (STG/NCs)***, their notations and semantics. To distinguish them from STGs described in earlier chapters, we call the latter ***STGs with input choices (STG/ICs)***. The behavior of STG/NCs will also be defined in terms of firing sequence semantics. From the behavioral standpoint, a STG defines a set of firing sequences which has an equivalent state graph representation. An STG can thus be viewed as an abstract representation of a state graph.

In Section 2 we will consider the underlying idea of specifying non-input choices in terms of state graph representations. In particular, we discuss the deficiency of state graphs for modeling non-input choices and a simple solution to this problem. This leads to a notational extension to the STG model, as described in Section 3. This section also describes the syntactic definition of STG/NCs and their firing rule. In Section 4, a design example is carried through to informally illustrate the main ideas. This example describes the synthesis of a FIFO controller which operates using a two-cycle signaling protocol. One unusual feature of this design is that its STG specification contains both concurrency and internal choices, in contrast to other synthesis approaches which restrict the specification to sequential processes, and thus cannot specify both internal choice and concurrency at the same time.

8.2 The Basic Idea

In this section, through a number of illustrative examples, we demonstrate a fundamental difficulty in modeling internal choices in the class of finite automata corresponding to state graphs. After identifying the cause of this problem, we suggest a solution which necessitates the use of a higher level representation for state graphs—this being STGs with non-input choices, as described subsequently.

8.2.1 A fundamental problem with specifying non-input choices in state graphs.

Recall that a state graph Φ defined on a set of signals J is given by $\Phi = \langle S, T, \delta, s_0 \rangle$ where $S = \{s : J \rightarrow \{0,1\}\}$ and $T = J \times \{+,-\}$. J and T can be partitioned into *input, internal* and *output* sets, denoted by subscripts I, N and O, respectively. For convenience, we also define *non-input* sets as the union of internal and output ones, and denote them with subscripts NI.

In Φ, suppose that there exists a configuration shown in Fig. 8.2a, with $s_1[u\rangle s_2$, $s_2[t_1\rangle s_3[t_3\rangle$, $s_2[t_2\rangle s_4[t_4\rangle$, where u is the only transition enabled in state s_1. According to the previous definition of enabling and disabling, this situation corresponds to uEt_1 and uEt_2 in s_1. Since t_1 and t_2 are enabled in the same state and $t_2 \notin T_E(s_3)$, $t_1 \notin T_E(s_4)$, they correspond to transitions in direct conflict. (Recall that $T_E(s)$ denotes the set of transitions which are enabled in state s.)

As argued below, whether t_1 and t_2 are indeed in direct conflict, that depends on whether they are transitions of input or non-input signals. For the sake of argument, let us consider the direct conflict in Fig. 8.2a purely as a *specification* intended for specifying a nondeterministic choice in some control module. Due to the fact that in a state graph, transitions behave differently depending on their *type* (input or non-input), whether this specification is in fact a choice that will depend on the type of the transitions. For state graphs, transitions of input signals are caused by the environment, whereas non-input ones are caused by the system itself. The occurrence of the latter is determined solely by the internal delays of the system and hence, cannot be known exactly. Below, these cases are examined.

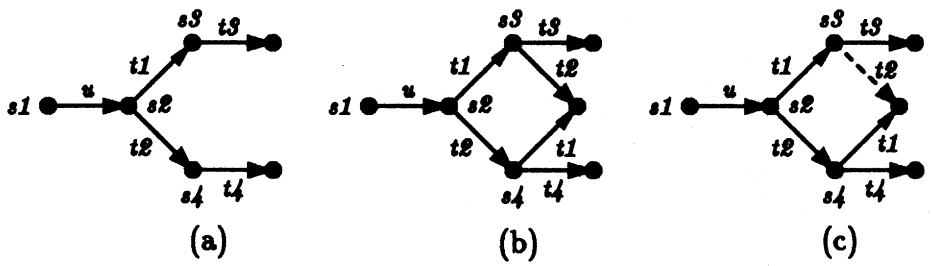


Figure 8.2: (a) A state graph specification with t_1 and t_2 enabled in the same state. (b) If $t_1, t_2 \neq \bar{u}$ then t_1 and t_2 are concurrent. (c) If $t_1 = \bar{u}$ then t_2 is non-persistent.

If $t_1, t_2 \in T_I$ then as discussed in Chapter 5, it is an *input* choice. Furthermore, t_1 and t_2 can be assumed to be persistent; their persistency can be guaranteed by the environment. On the other hand, if $t_1, t_2 \notin T_I$ then the specification of Fig. 8.2a doesnot guarantee that t_1 and t_2 will be persistent or that they may occur concurrently, as explained below.[1]

- $t_1, t_2 \neq \bar{u}$: Since uEt_1 in s_1 and $t_2 \neq \bar{u}$, the occurrence of t_2 in s_2 brings the system to s_4 in which t_1 must still be enabled: $s_4[t_1\rangle$. Similarly, since uEt_2 and $t_1 \neq \bar{u}$, we can also conclude that $s_3[t_2\rangle$. Hence $t_1, t_2 \notin T_I \cup \{\bar{u}\} \Rightarrow s_2[t_1\rangle s_3[t_2\rangle \wedge s_2[t_2\rangle s_4[t_1\rangle$, and this means that t_1 and t_2 are in fact concurrent (Fig. 8.2b).
- $t_1 = \bar{u}$ or $t_2 = \bar{u}$: Suppose that $t_1 = \bar{u}$ and $t_2 \neq \bar{u}$ (the other case with $t_1 \neq \bar{u}$ and $t_2 = \bar{u}$ can be treated in a similar fashion). As before, since uEt_1 in s_1 and $t_2 \neq \bar{u}$, t_1 must still be enabled in s_4. However, since uEt_2 in s_1 but $t_1 = \bar{u}$, the occurrence of t_1 in s_2 brings the system to state s_3 in which t_2 is no longer enabled (Fig. 8.2c). Hence, t_1 disables t_2 in s_2 thus causing t_2 to be non-persistent. The obvious implication of this is that in the corresponding hardware implementation, an occurrence of t_2 can cause a hazard which may lead to malfunction.

The above analysis shows that state graphs can only be used to specify *input choices*. If a specification involves non-input transitions enabled in the same state, due to the fact that their occurrences are controlled internally, these transitions will occur concurrently or produce hazards—neither case is desirable.

[1] We do not consider cases with $t_1 \in T_I \wedge t_2 \notin T_I$ or $t_2 \in T_I \wedge t_1 \notin T_I$ because these involve choices between input and non-input signal transitions.

8.2.2 Specifying non-input choices in state graphs.

Consider the state in Fig. 8.3a. Since x and y are enabled in the same state s and they are transitions of input signals (input transitions are underlined), they indeed specify an input choice. In contrast, t_1 and t_2 are enabled in the same state s_2, but since they are transitions of non-input signals, they may be non-persistent or concurrent, as discussed earlier. Hence, this state graph cannot be used to specify a non-input choice as intended.

Now, suppose that once transition u has occurred and the system has settled in state s_2, we want the system to perform the following decision:

- enable transition t_1 only if x had occurred previously, and
- enable transition t_2 only if y had occurred previously.

It can be seen that for the state graph in Fig. 8.3a, this type of decision is impossible because as soon as the system reaches state s_1, it looses the knowledge of which of x or y had occurred. In order to allow for non-input choices, we must "split" states s_1 and s_2 into pairs $\{s_1', s_1''\}$ and $\{s_2', s_2''\}$, respectively (Fig. 8.3b). In this state graph, once transition u has occurred, the enabling of t_1 is conditional on the occurrence of x and likewise, that of t_2 on y. Furthermore, by splitting state s_2, t_1 and t_2 are no longer enabled in the same state; the problem discussed above no longer exists. Thus this simple technique provides not only a way for specifying non-input choices but also a nice solution to the above problem.

As discussed next, the proposed solution gives rise to another problem, thus indicating a fundamental deficiency of state graphs in specifying non-input choices. Suppose that one is given the state graph in Fig. 8.3b. By noticing that uEt_1 in s_1' and uEt_2 in s_1'', one may be able to deduce that t_1 and t_2 are possible internal choices after the occurrence of u. However, in general, the exact condition which causes a particular choice cannot be determined precisely: by inspecting only the state graph, the condition which causes the choices of t_1 may be due to the occurrence of either one of x or t_5; similarly that t_2 may be due to either one of y or t_6.

In order to specify the exact conditions for non-input choices, new notations need to be introduced, as illustrated in Fig. 8.3d. The output arcs of place p are labeled with control variables a and b where, for instance, $a = \langle i, 0\rangle$, $b = \langle i, 1\rangle$, $x = i_-$ and $y = i_+$. In this case,

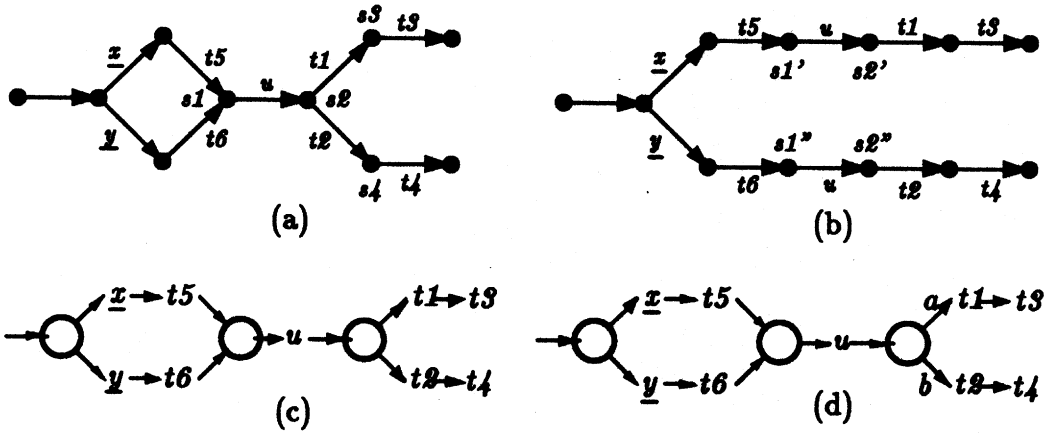


Figure 8.3: (a) A state graph intended for specifying an internal choice between t_1 and t_2. (b) The correct way for specifying internal choice between t_1 and t_2. (c) The STG for the state graph in (a). (d) The STG for the state graph in (b).

for example, when x fires, a becomes true and t_1 will be enabled. This STG can be viewed as a high-level representation of the state graph in Fig. 8.3b, and can be distinguished from the STG without these labeled arcs shown in Fig. 8.3c.

Before giving a precise syntactic characterization of STGs with non-input choices, the following remarks are in order.

- The use of arc labels in STGs allows one to specify non-input choices and the exact conditions which control the choices; such a capability is not supported by free-choice nets. An immediate implication of this extension is directly related to the hardware implementation: in the decomposition stage for logic implementation, signals corresponding to these conditions must also be included as input to a logic element. For example, in Fig. 8.3d signal i must be considered as an input to logic elements j and k, where $t_1 = j_*$ and $t_2 = k_*$.
- The controlled-choice notation has significant ramifications in our formulation of STGs in that it allows us to restrict the number of appearance of every transition in T to no more than once in a STG specification, and this is consistent with our previous formulation. (Recall from Chapter 2 that this restriction is necessary to prevent the confusion between concurrency and nondeterminism in the corresponding state graph representation.)
- In comparison with free-choices, controlled choices can be viewed as a further restric-

tion on the sets of firing sequences. For example, the set of firing sequences for the free-choice case in Fig. 8.3c is $\{xt_5ut_1t_3, xt_5ut_2t_4, yt_6ut_1t_3, yt_6ut_2t_4\}$, whereas that for the controlled choices in Fig. 8.3d is restricted to the subset $\{xt_5ut_1t_3, yt_6ut_2t_4\}$.

8.3 STGs with non-input choices

8.3.1 Syntax

A STG with non-input choices (STG/NCs) of a control module with a set of signals J is defined as follows.

Definition 8.1 *A STG with non-input choices defined on J is described by $\Sigma_J = \langle P, T, F, M_0; \lambda \rangle$, where*

- *$\langle P, T, F, M_0 \rangle$ is a LSFC net satisfying the one-token SM restriction, with the addition of a finite set of* dummy *transitions denoted by $\mathcal{E}$: $F \subseteq (P \times (T \cup \mathcal{E})) \cup ((T \cup \mathcal{E}) \times P)$.*
- *The* arc labeling *function $\lambda : F \rightarrow J \times \{0,1\}$ is a partial function with*

$$dom(\lambda) = \{\langle x, y \rangle \in F \mid x \in P \wedge |x \cdot| > 1 \wedge (y \in T_{NI} \cup \mathcal{E})\},$$

i.e., $dom(\lambda) \subseteq F \cap (P \times (T_{NI} \cup \mathcal{E}))$.

The set $\mathcal{E} = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_m\}$ is a finite set of dummy transitions, each member of $\mathcal{E}$ is really a *silence transition* whose purpose will be discussed shortly.

In the graphical representation of arc labels in $J \times \{0,1\}$, we use j and $\bar{j}$ to denote $\langle j, 1 \rangle$ and $\langle j, 0 \rangle$, respectively. An arc label $\langle j, 0 \rangle$ represents a control variable which holds whenever signal j is equal to 0.

We further require that for a place $p \in P$ such that $|p \cdot| > 1$,

- either $p \cdot \subseteq T_I$ (in which case p is called a *free-choice* place)
- or $p \cdot \subseteq T_{NI} \cup \mathcal{E}$ (in which case p is called a *controlled-choice* place).

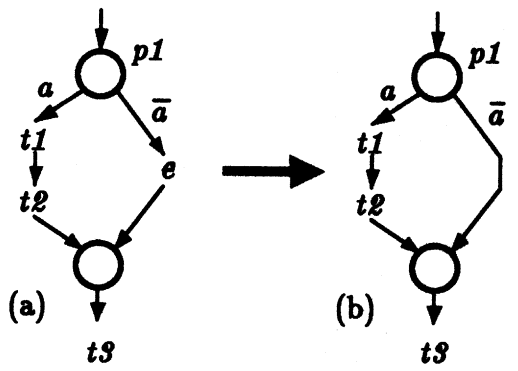


Figure 8.4: (a) A STG with non-input choices and (b) abbreviating the dummy transition.

Thus a STG/NC may contain both free choice and controlled-choice places, while a STG/IC may contain only free-choice places.

The introduction of the dummy transition ϵ is a mere technicality: in situations involving non-input choices (as illustrated in Fig. 8.4a), when p_1 is marked, if control variable a holds then t_1, t_2, t_3 occur in sequence; otherwise, if $\bar{a}$ holds then only t_3 will occur (the exact meaning of "hold" will be given shortly). The dummy transition ϵ serves as a place holder to avoid too many changes to the syntax of STGs and their firing rule: without it, we must extend F to $P \times P$ and change the firing rule accordingly. For convenience we will draw an arc directly between two places whenever an ϵ-transition is encountered, as done in Fig. 8.4b. Thus, the use of dummy transitions is restricted to the following cases:

$$\forall \langle x, y \rangle \in F \text{ if } y \in \mathcal{E} \text{ then } \begin{cases} \exists a \in J \times \{0,1\} : \lambda(\langle x, y \rangle) = a \text{ and} \\ \exists p_1, p_2 \in P : |p_1 \cdot| > 1 \wedge \cdot y = \{p_1\} \wedge |\cdot p_2| > 1 \wedge y \cdot = \{p_2\} \end{cases}$$

8.3.2 Firing rule

The firing rule for STG/NC is exactly the same as for Petri nets, i.e., whenever a transition (including the dummy transition ϵ) is enabled, its firing will remove one token from each of its input places and one token is added to each of its output places. However, the *enabling condition* for STG/NC is slightly different:

- For every transition t with *no* labeled input arcs, t is enabled the usual way: $M[t\rangle$ iff $\forall p \in \cdot t, M(p) \geq 1$.

- For every transition t with the input arc labeled with a, e.g. $\cdot t = \{p\}$ and $\lambda(p,t) = a$,[2] then $M[t\rangle$ iff (i) $\forall p \in \cdot t, M(p) \geq 1$ and (ii) the condition a ***holds*** at M.

We say that condition a ***holds*** at marking M

- for the case $a = \langle j, 0\rangle$ iff at M, transition j_- has fired ***and*** j_+ has not;
- for the case $a = \langle j, 1\rangle$ iff at M, transition j_+ has fired ***and*** j_- has not.

In other words, if s is the binary state corresponding to marking M, then the holding of a at s means that $s(j) = 0$ for the case $a = \langle j, 0\rangle$, and $s(j) = 1$ for $a = \langle j, 1\rangle$.

The above rules should provide an adequate recipe for determining the sets of firing sequences and their equivalent state graphs from STG/NCs. This is accomplished by determining at every marking the enabled transitions and by firing them to reach a new marking, and so on. However, in order to use them effectively, we need to enhance the analysis power for this type of STGs. Fortunately, a STG/NC can be converted into one containing only free-choice places by first unfolding it into an *occurrence net* and then folding it back into a free-choice net. Such procedures constitute the main steps of an ***expansion algorithm*** described in the next chapter. For free-choice nets, the results developed earlier in the thesis can be used directly. The presentation of this expansion algorithm necessitates the introduction of occurrence nets and processes generated from Petri nets, as will be discussed later. Below, we illustrate the use of STG/NCs with a design example of a FIFO controller, which uses a ***two-cycle*** handshake protocol for external communication.

8.4 An example: a two-cycle FIFO controller

We discuss an example of a STG specification of a two-cycle FIFO controller. The main objective of this example is to show the expressive power of this extended STG model and the type of asynchronous circuits that can be synthesized from it. To provide some intuition, an informal description of the expansion algorithm mentioned above will follow. The synthesis steps to produce the circuit realization will also be outlined.

[2]From now on, we write $\lambda(x,y)$ for $\lambda(\langle x,y\rangle)$.

The block diagram of the FIFO module is shown in Fig. 8.5a, a timing diagram describing its operation in Fig. 8.5b. The FIFO module has three control links, each being a pair of *request/acknowledge* signals. At the input link, R_i makes a transition each time new input data is available; A_i makes a transition each time the data has been used and the module is ready to accept new data. At the output link, R_o makes a transition whenever the module has stored data, and A_o makes a transition whenever the succeeding stage has accepted the data. Link $\{L, D\}$ is connected to a flipflop which simulates the data storing operation. Signal L makes a transition to load data into the registers, signal D makes a transition when the loading operation is done. For every cycle of operation, the signal L serves as a *load pulse* to control the input gates of register cells. This FIFO cell operates in a pipelined fashion by coordinating transitions at the input and output links. In this STG specification of the FIFO module, transitions of signals at the input link $\{I_r, I_a\}$ can occur concurrently with those at the output link $\{O_r, O_a\}$.

The STG specification for this circuit is shown in Fig. 8.5c, in which the output arcs of places p_1 and p_3 are labeled with variables D and R_i, respectively, and there are no free-choice places. Formally, the labeling function λ is defined by

$$\lambda(p_1, A_{i-}) = \langle D, 0\rangle \quad \text{and} \quad \lambda(p_1, A_{i+}) = \langle D, 1\rangle$$
$$\lambda(p_3, D_+) = \langle R_i, 1\rangle \quad \text{and} \quad \lambda(p_3, D_-) = \langle R_i, 0\rangle.$$

The set $\{D, R_i\}$ can be used to form the control states which dictate the choice between alternate control sequences.

In the initially marking $M_0 = \{p_5, \langle A_{i-}, R_{i+}\rangle\}$, transition R_{i+} is enabled. The initial state of the system is $s_0 = 000000$, where states are vectors of binary values of signals in $\langle R_i, A_i, L, D, R_o, A_o\rangle$. After the firing of R_{i+} and subsequently, L_+, place p_3 is marked. At this marking, the condition $\langle R_i, 1\rangle$ holds (because R_{i+} has just fired), thus enabling D_+. After D_+ has fired, L_- may fire concurrently with the sequence R_{o+}, A_{o+}, marking places p_1 and p_5. At this marking, A_{i+} is enabled due to the holding of the condition $\langle D, 1\rangle$. Thus one can "execute" the net and expand it into one shown in Fig. 8.5d. In this figure, there are two instances of L_+ denoted by L_+^0 and L_+^1 and two instances of L_-, likewise denoted. The finite automaton derived from this new STG is given in Fig. 8.5e, from which it can be seen that there is no state at which both A_{i+} and A_{i-} (similarly, D_+ and D_-) are enabled. Hence, by using arc labels to perform decisions, we have in effect "split" states which could have otherwise involved conflicts between transitions of non-input signals D and A_i.

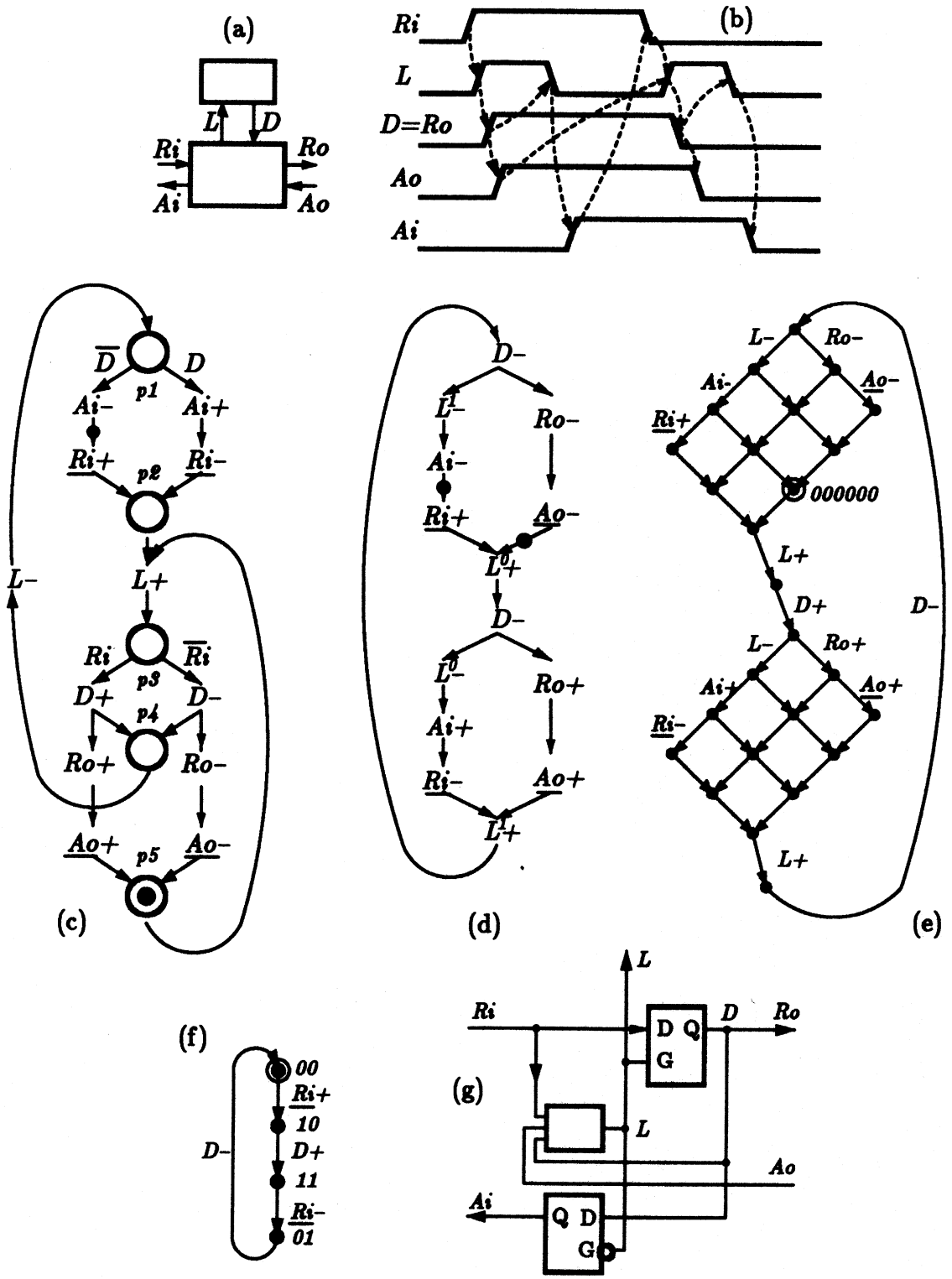


Figure 8.5: (a) The block diagram of the two-cycle FIFO controller, (b) its timing diagram, (c) its STG/NC specification which can be expanded into a STG/IC (d). (e) The state graph of the STG/IC in (d). (f) The control-state graph. (g) The final logic implementation, with $L = DA_o\overline{R}_i + \overline{D}\bar{A}_oR_i$.

It is useful to construct from Fig. 8.5d a state graph which consists only of control variables R_i, D. Such a state graph (Fig. 8.5f) is called a *control-state graph* (CSG) and is obtained by performing T'-contraction on the state graph in Fig. 8.5e, where $T' = \{R_i, D\} \times \{+, -\}$. A CSG contains only control states and it allows one to tell which choice is made at free- and controlled-choice places in the STG.

From the STG in Fig. 8.5d, the synthesis procedure described earlier can be applied directly to produce the logic equations and structure of the circuit. The only minor difference is that we need also to include the control variable D in the input set of A_i, and R_i in the input set of D. The input sets are given below, from which the structure of the logic circuit can be determined.

$$\begin{aligned} I(A_i) &= \{L, D\} \\ I(L) &= \{R_i, A_o, D\} \\ I(D) &= \{L, R_i\} \\ I(R_o) &= \{D\}. \end{aligned}$$

The circuit diagram for the FIFO cell is shown in Fig. 8.5g. There are two latches which pass the input D to output Q if the gating signal G is active, and hold the output if G is inactive. Their logic functions are $Q = DG + Q(\overline{G} + D)$. The logic function for signal L is $L = DA_o\overline{R}_i + \overline{D}\bar{A}_oR_i$. In a practical design, it is intended that the delay of the latch whose output is D be used to "time" the latching operation of the data registers for each stage of the FIFO. This circuit is completely speed-independent and has no problem with hazards, including the type called delay-hazards, under the following assumptions:

- Logic gates have unbounded delays, wires have no delays.
- The internal feedback delays of the latches are negligible compared to other loop delays.

Remarks. In the STG/NC (Fig. 8.5c) non-input choices are specified using controlled-choice places p_1 and p_3. These illustrate an important underlying mechanism when one considers STGs as a high-level representation of state graphs, which is the following:

- For a *free-choice* place, e.g. place p_0 with $p_0{\cdot} = \{x, y\} \subseteq T_I$ in Fig. 8.3c, in the state graph, there exists a state in which both x and y are enabled (Fig. 8.3a). This represents an *input choice*.

- For a *controlled-choice* place, e.g. place p_1 with $p_1{\cdot} = \{A_{i-}, A_{i+}\}$ in Fig. 8.5c, in the state graph (Fig. 8.5e), there exists *no* states in which both A_{i-} and A_{i+} are enabled. Thus a controlled-choice place indicates that its corresponding state in the state graph must be split so that each non-input choice is enabled in a different state. This is carried out by *expanding* the net to one shown in Fig. 8.5d, in which all non-input choices have been determined and removed by unfolding the original net. In the state graph, the states in which A_{i-} and A_{i+} are enabled (say s and s', respectively) are guaranteed to be different because the value of signal D is chosen such that it is different in these states: $s(D) = 0$ and $s'(D) = 1$. Hence the use of arc labels ensures that split states have different binary representations.

Chapter 9

The Expansion Algorithm

In this chapter, we describe an algorithm for converting STGs with non-input choices (STG/NCs) into STGs with input choices only (STG/ICs). This algorithm involves unfolding a STG/NC into a *process* and then folding the latter back into a STG/IC—a procedure called *expansion*. The underlying idea of the expansion algorithm can be understood by viewing STGs as high-level representations of state graphs. At the end of Chaper 8, we have emphasized that for a free-choice place, there exists a state with input choice, and for a controlled choice place, there exists a state with non-input choice. By manipulating the STGs, one can manage to split states with non-input choices in their corresponding state graphs. Specifically, this is carried out in two stages. In the first stage, *all* states with input *and* non-input choices are split; this is done by unfolding the STG/NC into a process. In the second stage, only states which previously involve input choices are merged back together; this corresponds to folding the process back into a STG/IC by merging only free-choice places.

This chapter is organized as follows. In Section 1, we describe yet another type of primitive nets called *occurrence nets*, which are commonly used for describing the semantics of higher level nets. In particular, the "execution" of live-safe Petri nets basically unfolds them into occurrence nets called *processes*. We will develop a number of useful properties for processes of nets and use them to derive some important results for processes of STGs. These results are required in the expansion algorithm described in Section 2. In this section, algorithms for net *unfolding* and *folding* of LSFC nets are given in order to motivate the development of the expansion algorithm which converts a STG/NC into a STG/IC.

Once a STG/IC is obtained, the synthesis procedure described in earlier chapters can be applied directly, as illustrated in an example at the end of Section 2. In Section 3, we study properties of STG/NCs, including liveness and the characterization of the temporal relation in these nets. Finally, Section 4 summarizes the main ideas and provides some remarks concerning our techniques.

It is important to note that this extension to the theory of Petri nets is a part of structure theory and hence, all results are guaranteed to apply only to nets which satisfy the one-token SM restriction stated earlier.

9.1 Occurrence nets and Processes of nets

In this section, we introduce a type of acyclic net structures called *occurrence nets*, being partial orders whose elements are *conditions* and *events*, together with a binary relation specifying the precedence relationship between these elements. Occurrence nets can be used as semantics for Petri nets: by executing a Petri net, one can unfold it into an occurrence net, and each execution produces an occurrence net called a *process*.

As argued earlier, for our purpose of using nets for synthesis of control systems, semantics based on firing sequences are more useful than those based on partial orders. Our chief motivation for presenting occurrence nets and processes is to provide a technique for net unfolding, which is part of the expansion algorithm described later.

9.1.1 Occurrence nets

Consider a type of elementary net structures represented by $N = \langle B, E, H \rangle$, where

- B is a set of *conditions* (similar to places and depicted likewise),
- E is a set of *events* (similar to transitions and depicted likewise),
- $H \subseteq E \times B \cup B \times E \neq \emptyset$ is the *flow relation*,

subject to the following restrictions:

- $E \subseteq dom(H) \cup range(H)$ (no isolated events) and

- $\forall b_1, b_2 \in B : (\cdot b_1 = \cdot b_2) \wedge (b_1 \cdot = b_2 \cdot) \Rightarrow b_1 = b_2$ (the net is pure).

Then N is an *occurrence net* iff

- $\forall b \in B : |b \cdot| \leq 1 \wedge |\cdot b| \leq 1$.
- $\forall x, y \in B \cup E : x H^+ y \Rightarrow \neg(y H^+ x)$ where H^+ denotes the transitive closure of H (H^+ is irreflexive).

For N, we also define the sets of *boundary* elements as follows:

$$\begin{aligned} {}^\circ N &= \{x \in B \cup E \mid \cdot x = \emptyset\}, \\ N^\circ &= \{x \in B \cup E \mid x \cdot = \emptyset\}. \end{aligned}$$

From an occurrence net, one can define a partial order[1] $\langle X, H^+ \rangle$ where $X = B \cup E$. Ordering and concurrency (unordering) between elements in X can then be defined as follows. For every $x, y \in X$, x and y are

- *ordered*, denoted as $\{x, y\} \in li$ or $x \, li \, y$, iff $(x H^+ y) \vee (y H^+ x)$;
- *concurrent*, denoted as $\{x, y\} \in co$ or $x \, co \, y$, iff $\neg(x H^+ y) \wedge \neg(y H^+ x)$.

The notions of *chains, lines, antichains* and *cuts* can be defined as follows.

- $l \subseteq X$ is a *chain* iff $\forall x, y \in l : x \, li \, y$; a *line* is a maximal chain.
- $c \subseteq X$ is an *antichain* iff $\forall x, y \in c : x \, co \, y$; a *cut* is a maximal antichain.

Also, *b-cuts* and *e-cuts* denote cuts consisting only of elements in B and E, respectively. Note that for LSFC nets, we use **li** and **co** to denote ordering and concurrency relations; these are characterized directly on LSFC nets, in contrast to the above relations which are defined on occurrence nets.

[1] A partial order is defined here as a set together with an *irreflexive* and transitive binary relation.

9.1.2 Processes of nets

Informally, a process of a Petri net is an occurrence net obtained by unfolding a Petri net into an acyclic structure involving only ordering and concurrency but no conflicts. For example, given a Petri net which is a simple cycle $p_1 t_1 p_2 t_2 \ldots p_n t_n p_1$ with p_1 initially marked, by unfolding this net one obtains an occurrence net corresponding to a *line:*

$$p_1^0 t_1^0 p_2^0 t_2^0 \ldots p_n^0 t_n^0 p_1^1 t_1^1 p_2^1 t_2^1 \ldots p_n^1 t_n^1 p_1^2 \ldots$$

where superscripts are used to denote instances of places or transitions. Even though nets can generate infinite processes, for all practical purposes, finite processes are adequate. Hence unless explicitly stated otherwise, we consider only finite processes. Formally, a process of a net is defined as follows [42].

Definition 9.1 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a Petri net and $N = \langle B, E, H \rangle$ an occurrence net with ${}^\circ N$, $N^\circ \subseteq B$. Then $PN = \langle B, E, H; \phi \rangle$ is a process of Σ iff $\phi : B \cup E \to P \cup T$ is a* surjective *function satisfying the following conditions (ϕ is extended to $\phi : \mathcal{P}(B \cup E) \to \mathcal{P}(P \cup T)$ in the obvious way):*

(a) $\phi(B) \subseteq P \wedge \phi(E) \subseteq T$.

(b) $\forall e \in E, \forall t \in T : \phi(e) = t \;\Rightarrow\; \phi(\cdot e) = \cdot t \wedge \phi(e\cdot) = t\cdot$.

(c) $\phi({}^\circ N) \subseteq \{p \in P \mid M_0(p) = 1\}$.

(d) $\forall x_1, x_2 \in B \cup E : \phi(x_1) = \phi(x_2) \Leftrightarrow x_1 \, li \, x_2$.

Condition (a) states that the mapping is type preserving: conditions map to places, events map to transitions. Condition (b) further states that if an event e maps to a transition t, then e's input (output) conditions map to t's input (output) places. Condition (c) requires that ${}^\circ N$ map to a subset of initially marked places. Finally, condition (d) states that two instances in the occurrence net map to the same element in the Petri net iff they are ordered; this is an important point to which we will return shortly.

Usually, we let $E \subseteq T \times \{0, 1, 2, \ldots\}$, so that events in E correspond to instances of transitions in T. For $t \in T$, its instances in E are denoted by $t^0, t^1, t^2, \ldots$ and hence

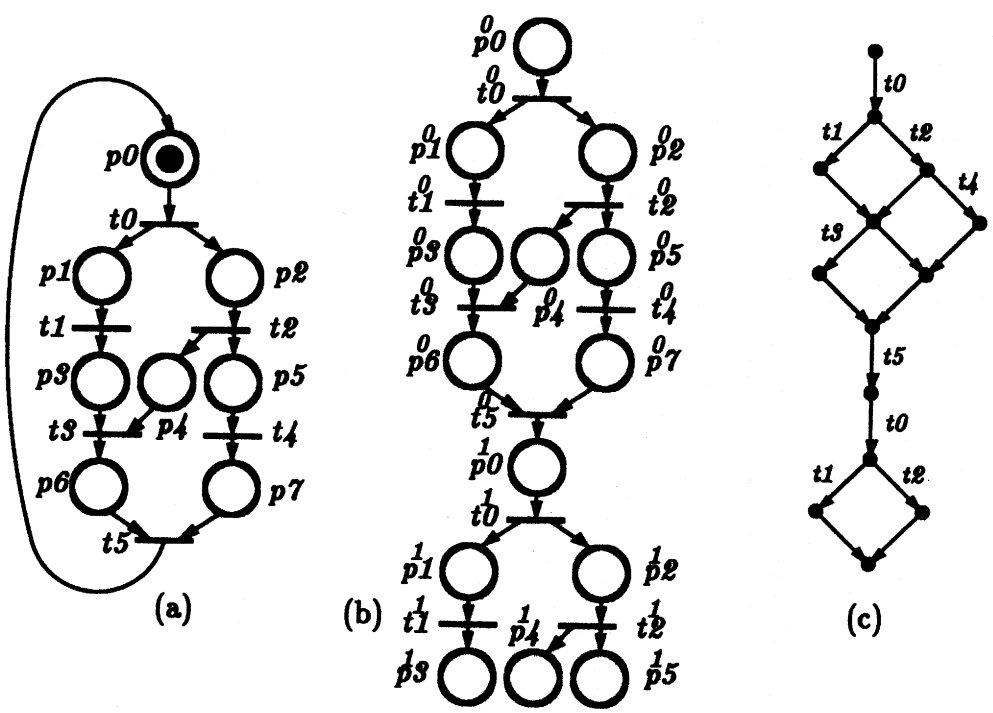


Figure 9.1: (a) A LS marked graph and (b) one of its processes. (c) The finite automata representation derived from the trace set of (b).

$\phi(\{t^0, t^1, t^2, \ldots\}) = \{t\}$. Similarly, we let $B \subseteq P \times \{0, 1, 2, \ldots\}$, so that conditions corresponding to instances of a place p are denoted by $p^0, p^1, p^2 \ldots$ and hence $\phi(\{p^0, p^1, p^2, \ldots\}) = \{p\}$. Processes always start and end with b-cuts: ${}^\circ N, N^\circ \subseteq B$, as shown in Fig. 9.1b.

The notion of process is closely connected to the execution of live-safe mark graphs. In fact, a LS marked graph can be directly unfolded into a process without the omission of any transition during unfolding. Fig. 9.1a is a LS marked graph which can be unfolded into the process in Fig. 9.1b. This process possesses a certain periodicity: it repeats the same behavior whenever it reaches a b-cut which maps to the initially marked places in M_0.

9.1.3 A few results for processes of LSFC nets

Most of the results developed earlier for LS marked graphs can be applied to processes with some slight modification. As remarked at the beginning of this section, a live-safe Petri net which is a *simple cycle* unfolds into a *line*, as defined earlier. For LSFC nets, we define two elements of a net to be ordered iff they belong to a simple cycle. This means that in a process generated from a LSFC net, all ordered elements belong to the same line. Thus for LSFC nets, the above fact generalizes as follows.

Lemma 9.2 *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net satisfying the one-token Sm restriction and $PN = \langle B, E, H; \phi \rangle$ a process of Σ. Then for every $x, y \in P \cup T$:*

(a) *If $\{x, y\} \in \mathbf{li} \cup \mathbf{cf}$ then there exists a* line *in PN to which* all *instances of x and y, $x^i, y^j \in B \cup E$, belong.*

(b) *If $\{x, y\} \in \mathbf{co}$ then there exists* no line *in PN to which* all *instances of x and y, $x^i, y^j \in B \cup E$, belong.*

Sketch of proof. For part (a), if $\{x, y\} \in \mathbf{li}$ then this is a formal statement of the fact that a simple cycle unfolds into a line. If $\{x, y\} \in \mathbf{cf}$ then x, y must belong to a SM-component which is live-safe of Σ. This SM-component must also unfold into a line in the process PN as at any marking, exactly one of its places can be marked and no more than one of its transitions is enabled.

For part (b), if $\{x, y\} \in \mathbf{co}$ then there exists no simple cycle in Σ to which both x, y belong, and there exists a MG-component containing both. It is simple to verify that in this case, when the MG-component is unfolded, there cannot be a line containing all instances of x and y. ∎

From this lemma, we can further deduce that since every $x \in P \cup T$ belongs to the same simple cycle with itself, there exists a *line* in PN to which *all* instances of x belong. Another consequence of the above lemma is that a p-cut in a net corresponds to a b-cut in its process. Since a p-cut represents a set of places which can be marked concurrently and hence corresponds to a marking of a net, a b-cut is a record of the holding of a marking of the net. Similarly, there is a correspondence between t-cuts and e-cuts; the former representing a set of concurrent transitions, the latter a set of concurrent events.

Chapter 3 has provided a semantics of LSFC nets based on firing sequences. For the case of LS marked graphs, their sets of firing sequences can be obtained by weaving the sequences derived from a set of simple cycles which cover the nets. This result can be adapted directly to occurrence nets to allow the determination of their corresponding trace sets and equivalent automata. For example, the trace set of the process in Fig. 9.1b can be obtained by first choosing a set of *lines* which cover the structure, deriving their corresponding traces and weaving them. The equivalent automata representation for Fig. 9.1b is given in Fig. 9.1c. This idea can be formalized as follows. In an occurrence net $N = \langle B, E, H \rangle$, a set of lines $L = \{l_1, l_2, \ldots, l_n\}$ *covers* net N iff $\forall \langle x, y \rangle \in H, \langle x, y \rangle$ belongs to some line l_i in N. For a line $l_i = b_1 e_1 b_2 e_2 \ldots e_m b_m$,[2] its FA (denoted as FA_i) can easily obtained by considering *conditions* as *states* and *events* as *transitions* between states. Then similar to the construction algorithm in Chapter 3, the equivalent FA of a process N (denoted as FA_N) is given by

$$FA_N = FA_1 \parallel FA_2 \parallel \ldots \parallel FA_n.$$

9.1.4 Processes of STGs

Instead of considering processes generated from LSFC nets, we consider those generated from STGs. The main difference is that a process $PN = \langle B, E, H; \phi \rangle$ generated from an STG $\Sigma_J = \langle P, T, F, M_0 \rangle$ contains events which map to complementary pairs of signal

[2]Note that in a process, every line starts and ends with a condition in B.

transitions: $\forall t \in T\ \exists e_i, e_j \in E$ such that $\phi(e_i) = t$ and $\phi(e_j) = \bar{t}$. As mentioned earlier, a b-cut in a process corresponds to a marking of a net. For processes of STGs, such a b-cut has a corresponding binary representation s. Similar to the state-assignment function defined in Section 4.2, let $\alpha : \mathcal{P}(B) \rightarrow S$ be a mapping from b-cuts to binary states. This mapping provides a mechanism for establishing the correspondence between b-cuts in a process and binary states in the state graph. This mechanism is required for the following two purposes:

(a) In *unfolding* a STG/NC into a process, we may reach a b-cut which contains a controlled-choice place. At that point, we need to evaluate which control variable holds in order to proceed with unfolding the net. This mechanism allows one to determine the states of control variables from the process generated thus far.

(b) In *folding* a process into a net, we "merge" b-cuts in the process which correspond to the same binary state. Again, this mechanism permits one to determine whether two b-cuts have the same binary representation.

Below we consider these two cases. Due to the similarity between processes of nets and marked graphs, most of the results developed earlier for marked graphs and STGs can be applied to processes of STGs with only a slight modification. In the following, we state most of the results without proof and only appeal to their close relation with results for STGs.

The conditions on a process so that its equivalent FA has a consistent state assignment (c.s.a. for short) are similar to those for marked graphs. For brevity, in the following, we refer to a process or a net satisfying the conditions so that its equivalent FA has a c.s.a. simply as a process (or net) with a c.s.a.

For $t \in T$, let $E(t) \stackrel{def}{=} \{e \in E \mid \phi(e) = t\}$, the set of events which map to transition t. Then the process PN has a c.s.a. iff $\forall t \in T$, there exists a line l in PN such that

(*a*) *every element in* $E(t) \cup E(\bar{t})$ *belongs to* l, *and*

(*b*) *elements in* $E(t)$ *and* $E(\bar{t})$ *alternate in* l, *i.e.*,

$$\begin{array}{ll} \textit{either} & FS(l) \lceil E(t) \cup E(\bar{t}) = t^0\bar{t}^0t^1\bar{t}^1t^2\bar{t}^2\ldots, \\ \textit{or} & FS(l) \lceil E(t) \cup E(\bar{t}) = \bar{t}^0t^0\bar{t}^1t^1\bar{t}^2t^2\ldots. \end{array}$$

It has been shown in Theorem 4.8 that a STG has a c.s.a. iff each of its MG-components has a c.s.a. In such a STG, t and $\bar{t}$ are ordered for any pair of transitions $t, \bar{t}$. Thus when it is unfolded, all events corresponding to t and $\bar{t}$ must belong to the same line and they must alternate. It follows that if a STG has a c.s.a. then every process generated from it has a c.s.a. Furthermore, according to Theorem 5.2, a STG is live only if it has a c.s.a. Hence, if a STG is live then each of its processes has a c.s.a.

For distinct b-cuts B_i, B_j in PN, we also define the *interval* from B_i to B_j $(i < j)$ as

$$[B_i, B_j] \stackrel{def}{=} \{x \in B \cup E \mid x \textit{ belongs to a chain from } B_i \textit{ to } B_j\}.$$

Let s_i and s_j be binary states corresponding to B_i and B_j, respectively, i.e., $s_i = \alpha(B_i)$ and $s_j = \alpha(B_j)$. Then in the equivalent FA of the process PN, every path σ from s_i to s_j contains all events in $[B_i, B_j]$ and therefore $|\sigma| = |E \cap [B_i, B_j]|$.

An interval $[B_i, B_j]$ forms a *complementary set* iff

$$\forall t \in T: \ |[B_i, B_j] \cap E(t)| = |[B_i, B_j] \cap E(\bar{t})|.$$

The following lemma states that b-cuts B_i and B_j have the same binary representations iff the interval $[B_i, B_j]$ forms a *complementary set.*

Lemma 9.3 *Let Σ_J be a live STG and PN a process generated from it, as defined above. Then for every distinct b-cuts B_i, B_j in PN :*

$$\alpha(B_i) = \alpha(B_j) \Leftrightarrow [B_i, B_j] \textit{ forms a complementary set.}$$

Proof. If Σ_J is live, PN must have a c.s.a.. Let $s_i = \alpha(B_i)$ and $s_j = \alpha(B_j)$. If $[B_i, B_j]$ forms a complementary set then every path $\sigma : s_i[\sigma\rangle s_j$ in the state graph of PN must contain (i) the same number of events e and $\bar{e}$, where $\phi(e) = t$ and $\phi(\bar{e}) = \bar{t}$ for some $t \in T$ and (ii) they must alternate. In this case, it is simple to verify that $s_i = s_j$. ∎

Let Σ_J be a live STG and PN a process generated from Σ_J. Let B_0 denote ${}^\circ N$, then the initial state s_0 and B_0 are related by $s_0 = \alpha(B_0)$. The following lemma shows how to determine the binary state corresponding to any b-cut B_j in PN.

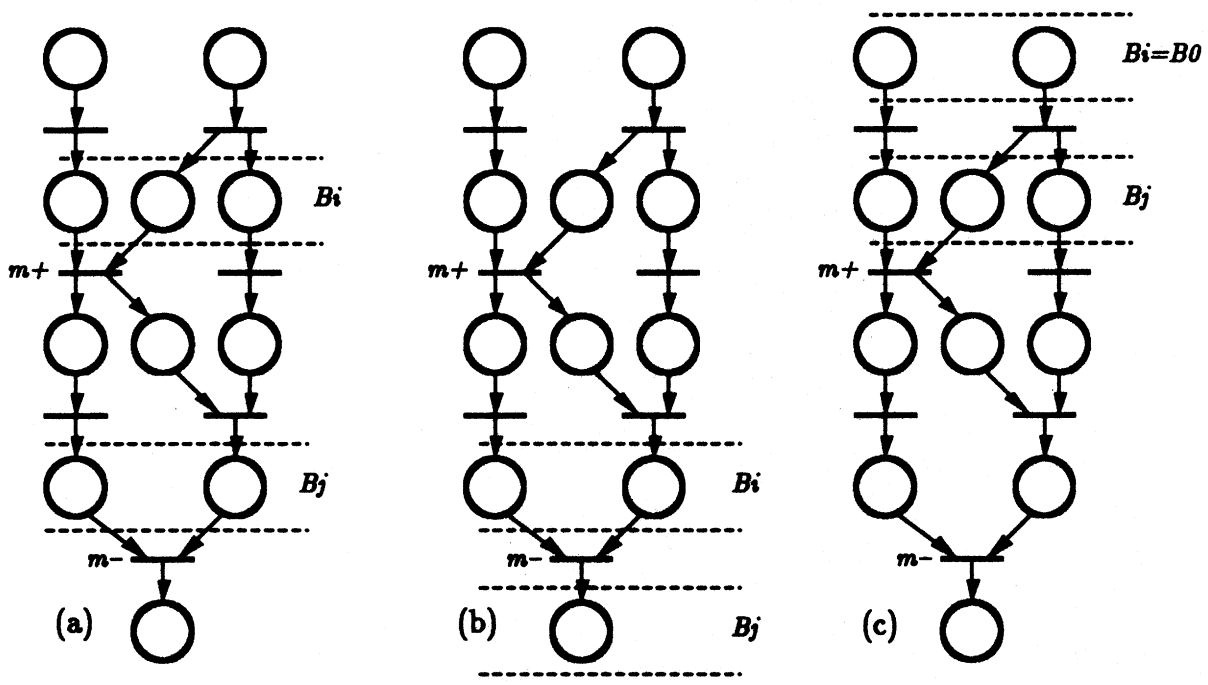


Figure 9.2: (a) At B_j, $s_j(m) = 1$. (b) At B_j, $s_j(m) = 0$. (c) At B_j, $s_j(m) = s_0(m)$.

Lemma 9.4 *Let $\Sigma_J = \langle P, T, F, M_0 \rangle$ be a live STG defined on the signal set J and $PN = \langle B, E, H; \phi \rangle$ a process of Σ_J. Then for every signal $m \in J$ and for every b-cut B_j in PN, the value of signal m at state $s_j = \alpha(B_j)$ is given by*

(a) $s_j(m) = 1$ *if there exists* $B_i \neq B_j$ $(i < j)$ *such that* $m_+ \in [B_i, B_j] \ \wedge \ m_- \notin [B_i, B_j]$.

(b) $s_j(m) = 0$ *if there exists* $B_i \neq B_j$ $(i < j)$ *such that* $m_- \in [B_i, B_j] \ \wedge \ m_+ \notin [B_i, B_j]$.

(c) $s_j(m) = s_0(m)$ *otherwise, where* $s_0 = \alpha(B_0)$, $B_0 = {}^\circ N$.

The proof is quite easy; it hinges on the fact that in the process PN of a live STG, for every transition $t \in T$ the subsets of events $E(t)$ and $E(\bar{t})$ belong to a line, and furthermore PN has a consistent state assignment. Fig. 9.2 illustrates these cases. Note that in case (c), $B_i = B_0$ and $[B_0, B_j]$ contains neither m_+ nor m_-. This means that signal m has not changed since the initial state s_0.

9.2 The expansion algorithm for STG/NCs

The expansion of STG/NCs consist of two algorithms: an unfolding and a folding one. First, we develop such a pair of algorithms for *FC nets*; these algorithms are then modified into the *expansion* algorithm for STG/NCs described subsequently.

9.2.1 Unfolding and folding of free-choice nets

Processes of live-safe Petri nets can be generated by simply unfolding the nets, resolving conflicts along the way. For free-choice nets, unfolding is a simple procedure which is intimately related to the MG-reduction algorithm due to Hack, described in Chapters 2 and 3. As before, for a Petri net $\langle P, T, F\rangle$, define an *allocation function* $A : P \to T$ which allocates an output transition for every place, i.e. $\forall p \in P : A(p) \in p\cdot$. In a free-choice net, if a place p has more than one output transition then any one of them can be allocated arbitrarily.

The following net unfolding algorithm unfolds the net iteratively and in each iteration, A is determined only for a *p-cut* corresponding to the last set of conditions of the process generated thus far. (In contrast, in the MG-reduction algorithm, A is determined for *all* places in P *at once.*) In the following, B_i is used to denote a set of *conditions* generated during a step of unfolding, and it is not necessarily a b-cut.

Algorithm 9.5 (Net Unfolding) *Let $\Sigma = \langle P, T, F, M_0\rangle$ be a free-choice net and $PN = \langle B, E, H; \phi\rangle$ a process of Σ. Then PN is constructed by iteratively unfolding Σ as follows.*

(*a*) *Initialization: Let $P_0 \subseteq \{p \in P \mid M_0(p) = 1\}$ such that P_0 is a p-cut.*
Let $i = 0$.

(*b*) *One step of unfolding consists of the following.*

b1. Define B_i such that $\phi(B_i) = P_i$ and $|B_i| = |P_i|$.

b2. Pick the allocation function $A(P_i)$ and define

$$T_i = \{t \in A(P_i) \mid (\forall p \in \cdot t)(\exists b \in \bigcup_{0 \le j \le i} B_j) : \phi(b) = p \wedge b\cdot = \emptyset\}.$$

b3. Define E_i such that $\phi(E_i) = T_i$ and $|E_i| = |T_i|$.

b4. Define $UH_i \subseteq (\bigcup_{0\leq j\leq i} B_j) \times E_i$ such that

$$(\forall b \in \bigcup_{0\leq j\leq i} B_j \mid b\cdot = \emptyset)(\forall e \in E_i) : \langle \phi(b), \phi(e)\rangle \in F \Leftrightarrow \langle b, e\rangle \in UH_i.$$

b5. Let $P_{i+1} = (T_i)\cdot$. Define B_{i+1} such that $\phi(B_{i+1}) = P_{i+1}$ and $|B_{i+1}| = |P_{i+1}|$.

b6. Define $DH_i \subseteq E_i \times B_{i+1}$ such that $\langle x, y\rangle \in DH_i \Leftrightarrow \langle \phi(x), \phi(y)\rangle \in F$.

b7. Let $H_i = UH_i \cup DH_i$.

(c) *Let $i = i+1$. Process PN can be further generated by going back to step (b), otherwise, it is given by*

$$B = \bigcup_{0\leq j\leq i} B_j, \quad E = \bigcup_{0\leq j<i} E_j, \quad H = \bigcup_{0\leq j<i} H_j.$$

It is important to note that step *b2* is most crucial in the algorithm, and means that for each place $p \in P_i$:

- If $|p\cdot| > 1$ (e.g. $p\cdot = \{t_1, t_2\}$) then $A(p)$ is chosen nondetermistically (e.g. $A(p) = t_2$). Since p is a free-choice place, t_2 has no other input places and it can be included in T_i.
- If $|p\cdot| = 1$ (e.g. $p\cdot = \{t\}$) then $A(p) = t$. In this case, however, t may have other input places (i.e., $\cdot t \geq 1$); t is included in T_i only if all *conditions* corresponding to $\cdot t$ have already occurred in the process generated thus far.

An example of net-unfolding applied to a LSFC net (Fig. 9.3a) is shown in Fig. 9.3b. Notice that only some of B_i's are b-cuts, some of E_i's are e-cuts.

Folding of a process of a LSFC net is rather simple and uninteresting; nevertheless, we describe it here to motivate the folding operation—similar in spirit—required for STGs with non-input choices.

Let Σ be a LSFC net and PN a process of Σ, obtained by applying the above unfolding algorithm. In order for PN to reproduce the original net Σ when folded, elements of PN must map to the set of *all* elements of Σ. In other words, the smallest process PN which can be folded back into Σ must satisfy

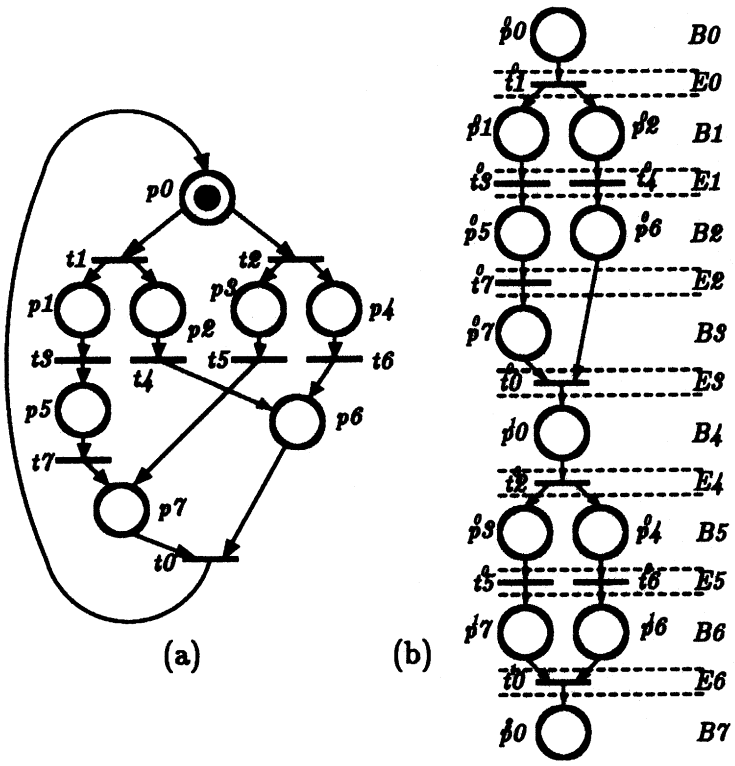


Figure 9.3: A demonstration of the net unfolding algorithm, as applied to a LSFC net (a) to produce a process (b).

(i) $\phi(^\circ N) = \phi(N^\circ) \subseteq \{p \mid M_0(p) = 1\}$ and

(ii) $B \cup E$ is the *minimal* set such that $\phi(B \cup E) = P \cup T$.

Such a process will be called *complete*. The following net folding algorithm is applied to complete processes of LSFC nets.

Algorithm 9.6 (Net folding) *Let $\Sigma = \langle P, T, F, M_0 \rangle$ be a LSFC net and $PN = \langle B, E, H; \phi \rangle$ a* complete *process of Σ. Let $N = \langle B, E, H \rangle$ be the associated occurrence net. Then*

$$\begin{aligned} P &= \phi(B), \\ T &= \phi(E), \\ F &= \{\ \langle \phi(x), \phi(y) \rangle \mid \langle x, y \rangle \in H\ \}, \\ M_0 &= \phi(^\circ N). \end{aligned}$$

It is a simple matter to check that by using the above definition, the process in Fig. 9.3b can indeed be folded back to the net in Fig. 9.3a. The reason why the folding of such a finite process reproduces the original LSFC net should be intuitively clear: when folded back, every place will have the same input and output transitions again. Likewise, every transition will have the same input and output places again; however, this is trivially true because by definition, a process preserves the input and output conditions of all events. This folding procedure reproduces the original net provided that the process contains at least one instance of every element of the original net.

We are now ready to discuss the expansion algorithm for producing a STG/IC from a STG/NC.

9.2.2 The Expansion Algorithm

Similar to the unfolding and folding algorithms developed for LSFC nets above, the expansion of STG/NCs also consists of two stages corresponding to unfolding and folding. First, we present an unfolding algorithm for STGs with input choices. This algorithm is identical to the one for free-choice nets given above (Algorithm 9.5), with the exception of step *b*2. In the previous unfolding algorithm, at every unfolding step, the allocation

function $A(P_i)$ is determined by randomly picking an output transition for every place $p \in P_i$ with more than one output transitions. For STG/NCs, a process can be generated in a similar manner; however, when a place $p \in P_i$ with labeled output arcs is encountered, an output transition is chosen depending on which arc label holds—the choice of which output transitions to pick is no longer free but dependent on a condition which can be determined from the process generated thus far. Below we will only describe the first part of step *b*2 for the new unfolding algorithm; this part involves the determination of the allocation function.

Algorithm 9.7 (Unfolding of STG/NC) *First part of step b2: Choose the allocation function $A(P_i)$ as follows.*

*b*2.1 *If $\exists p \in P_i$ such that p has no labeled output arcs then a transition $t \in p\cdot$ is picked randomly.*

*b*2.2 *Otherwise, if $\exists p \in P_i$ such that $p\cdot = \{t_1, t_2, \ldots, t_n\}$ and $\lambda(p, t_m) = a_m, 1 \leq m \leq n$, then $A(p) = t_m$ only if a_m holds at B_i.*

Important remarks. Case *b*2.1 is the same as one in the previous algorithm. For the new case *b*2.2, we have the following remarks.

- In order for the choice to be deterministic at B_i, it is required that exactly one a_m holds at B_i. Thus in the STG specification, the set $\{a_1, a_2, \ldots, a_n\}$ must be chosen such that every time place p is marked, only one condition in this set can become true. On the other hand, if it is possible for more than one of them, say a_i and a_j, to hold at B_i then both t_i and t_j are enabled. This situation will lead to problems as discussed in Chapter 8, concerning the specification of non-input choices.

  The above requirement ensures that every transition in $p\cdot$ is enabled in a marking with a distinct binary representation, as illustrated by the example at the end of Chapter 8.
- As mentioned at the beginning of Chapter 8, for STG/NC, we impose the restriction that

  $\forall p \in P, \forall t \in T$: *if* $\lambda(p, t) = a_m$ *then* $\{p, m_*\} \notin \mathbf{co}$, *where* $a_m \in \{\langle m, 0\rangle, \langle m, 1\rangle\}$. *That is, p cannot be concurrent with any transition of signal m.*

If this restriction is satisfied, then in the process PN of the STG, all instances of p, m_+ and m_- must belong to the same *line*. Thus by definition, in PN, any cut containing an instance of p cannot contain instances of m_+ or m_-. Therefore, if a_m holds at a b-cut B_i such that $p \in \phi(B_i)$, then a_m holds at any other b-cut B_i' such that $p \in \phi(B_i' \cap B_i)$. This is the reason why in step $b2$ above, it is sufficient to consider the holding of a_m at only one b-cut (instead of *all* b-cuts) containing p.

Conversely, without this restriction, then to determine whether a_m holds at p, one needs to check *every* b-cut B_i' such that $p \in \phi(B_i' \cap B_i)$. But in general, this is impossible because the process generated thus far may not contain all such b-cut B_i'. Therefore, this restriction is of fundamental importance; without it a STG/NC cannot be unfolded into a process.

By using the above algorithm, the STG $\Sigma_J = \langle P, T, F, M_0; \lambda \rangle$ can be iteratively unfolded into a process $PN = \langle B, E, H; \phi \rangle$. When the algorithm is stopped, $i = n$ for some positive integer n and

$$B = \bigcup_{0 \le j \le n} B_j, \quad E = \bigcup_{0 \le j < n} E_j, \quad H = \bigcup_{0 \le j < n} H_j.$$

Similar to the algorithms for free-choice nets, in order for the process generated to exhibit all possible behavior of the original STG when folded, we require that it be *complete* and has a consistent state assignment. That is, process PN must satisfy the following conditions which together constitute the *completeness of processes of STGs.*

Definition 9.8 (Complete Process of a STG) *Let $\Sigma_J = \langle P, T, F, M_0; \lambda \rangle$ be a STG/NC and $PN = \langle B, E, H; \phi \rangle$ a process of Σ_J. Let $N = \langle B, E, H \rangle$ be its associated occurrence net; $B_0 = {}^\circ N$ and $B_n = N^\circ$. Then PN is a* complete *process of Σ_J if the following conditions are satisfied:*

(*a*) $\phi(B_0) = \phi(B_n) \subseteq \{p \in P \mid M_0(p) = 1\}$.

(*b*) *$B \cup E$ is a minimal set such that $\phi(B \cup E) = P \cup T$.*

(*c*) *PN has a consistent state assignment and $[B_0, B_n]$ forms a complementary set.*

Note that (*a*) and (*b*) are the requirements for processes of free-choice nets to be complete, as discussed earlier. The last condition (*c*) implies that B_0 and B_n have the same binary representation, i.e., $\alpha(B_0) = \alpha(B_n)$.

The folding procedure consists of two steps. First, a complete process is folded by merging b-cuts B_0 and B_n to produce a strongly connected net. Then, free-choice places are merged, using the *recursive merge function* defined later. Controlled-choice places are *not* merged. The result is a free-choice net which behaves exactly like the original STG/NC; however, this new net can be synthesized using the techniques developed earlier.

Algorithm 9.9 (Folding of STG/NC) *Let $\Sigma_J = \langle P, T, F, M_0; \lambda \rangle$ be a STG/NC and $PN = \langle B, E, H; \phi ran$ a* complete *process generated from it using the unfolding algorithm described above. Then PN can be folded back into a net Σ'_J using the following procedure.*

(a) *Merge B_0 and B_n : $\forall b \in B_0, \forall b' \in B_n$ such that $\phi(b) = \phi(b')$: replace b' with b in B, E, H and ϕ.*

(b) *Merge free-choice places: For all $b, b' \in B$ such that $\phi(b) = \phi(b') = p \in P$ and p is a free-choice place,*
if there exist b-cuts $B_i, B_j, i < j$, such that $b \in B_i, b' \in B_j$ and $[B_i, B_j]$ forms a complementary set
then $merge(b, b')$.

(c) *The* expanded *net $\Sigma'_J = \langle P', T', F', M'_0 \rangle$ is given by*

$$\begin{array}{ll} P' = B & (\phi(P') = P) \\ T' = E & (\phi(T') = T) \\ F' = H & \\ M'_0 = B_0. & \end{array}$$

Given the process $PN = \langle B, E, H; \phi \rangle$, the *merge* function recursively merges elements of the process together and update all components B, E, H and ϕ of PN. Input to the merge function is a pair of elements (x_1, x_2) of the same type (conditions or events), they are merged into a single element named x_1.

Function $merge(x_1, x_2)$
    Replace all x_2 with x_1 in B, E, H and ϕ;
    For every y_1, y_2 such that $\phi(y_1) = \phi(y_2) \wedge (y_1\cdot = y_2\cdot \vee \cdot y_1 = \cdot y_2)$:
        $merge(y_1, y_2)$

Remarks. The above folding algorithm is the same as one for LSFC nets, except that only *conditions* corresponding to *free-choice* places are merged together, whereas those corresponding to *controlled-choice* places are not. This algorithm accomplishes the last stage of the expansion algorithm. The essential idea of the expansion algorithm is to split states with non-input choices. This is carried out in two stages: splitting *all* states with input *and* non-input choices and then merging states with input choices. The first stage corresponds to unfolding the STG/NC into a process; the second folding the process back into a STG/IC by merging only free-choice places.

In this algorithm, conditions in a process are merged only if (i) they map to the same free-choice place in the net *and* (ii) the b-cuts containing them must map to the *same* binary state. Requirement (ii) is unique to processes of STGs and it allows one to determine when conditions can be merged together. Specifically, this requirement appears in the following steps of the algorithm.

- In step (a), if PN has a c.s.a. then the binary states corresponding to the b-cuts B_0 and B_n must be identical: $\alpha(B_0) = \alpha(B_n)$. This implies that the state graph is strongly connected (and is therefore live).
- In step (b), if the b-cuts B_i and B_j are such that $[B_i, B_j]$ forms a complementary set and PN has a c.s.a., then they have the same binary representation: $\alpha(B_i) = \alpha(B_j)$. Hence in the state graph representation of PN, every event enabled in B_i will also be enabled in B_j and vice versa—this is exactly a free-choice situation. Thus, even though B_i and B_j appear to be separate b-cuts in PN, due to the fact that they have the same binary representation, they actually map to a *p-cut* containing the same free-choice place. It is important to realize that only by considering the binary representation of b-cuts, one can guarantee that they can be merged into free-choice places. If two b-cuts map to the same free-choice place but they do not have the same binary representation then they cannot be merged. An example illustrating these rules will be described next.

9.2.3 An example

In this section, we present another example of a control circuit with data-dependent operation which is used together with an arbiter to control access to shared resources. This is

called a Resource Locking Module (RLM) which communicates externally using the reset-signaling protocol. The reason for presenting this example is that its STG specification contains both free-choices and controlled choices, thus allowing a demonstration of the expansion algorithm just described. In the previous example of the two-cycle FIFO controller, its STG specification contains only controlled-choices. Hence, when the expansion algorithm was applied, no merging was needed; the net was unfolded into a process and then simply folded back at b-cuts corresponding to initially marked places.

Fig. 9.4a shows the block diagram of a circuit in which there are two RLMs connected to a two-input arbiter. Each RLM has a *set* link $\{S_r, S_a\}$, a *clear* link $\{C_r, C_a\}$ connected to the external environment, and a *lock* link $\{L_r, L_a\}$ connected to one port of the arbiter. When a set-request arrives at S_r, the RLM forwards it to L_r, awaits for acknowledgement on L_a and passes it back to S_a. This signifies that the arbiter has been locked. When the set-request S_r is dropped, the module immediately drops S_a in response, leaving L_r high to lock up the arbiter. From here on, any set-request will be acknowledged immediately through S_a. In order to unlock the arbiter, instead of a set-request, a clear-request C_r is sent to the RLM. This will reset L_r to low, which in turns will cause L_a to drop. The module responds by raising C_a, and when C_r is dropped, it will drop C_a in return, thus completing a *clear* cycle. A subtle timing requirement for the arbiter is that the input request at its other port is disallowed until signal L_a has gone low at this port. Such an aribter design is described in [10] and contains some subtle difference to that suggested by Seitz and also Plummer [40].

The STG specifying the operation of the RLM is shown in Fig. 9.4c; the logic implementation derived from the STG is shown in Fig. 9.4b. It can be easily verified that this circuit works according to the specification. In this STG, p_0 is a free-choice place; p_1 is controlled-choice place with arc labels $\langle L_a, 0\rangle$ and $\langle L_a, 1\rangle$. The variables controlling data-dependent operations are S_r, C_r and L_a. The control state graph is shown in Fig. 9.4g, where states are values of the vector $\langle S_r, C_r, L_a\rangle$. In the initial state s_0 of the system, the values of all signals in $\langle S_r, S_a, C_r, C_a, L_r, L_a\rangle$ are 0: $s_0 = 000000$.

This STG can be unfolded into a complete process which is a *line* because the system is totally sequential. As indicated in Fig. 9.4d, this *line* has four instances of place p_0, denoted by p_0^0, p_0^1, p_0^2 and p_0^3. For clarity, only these instances of place p_0 are drawn explicitly, other places are omitted from the figure. The line segment between p_0^0 and p_0^1 corresponds to

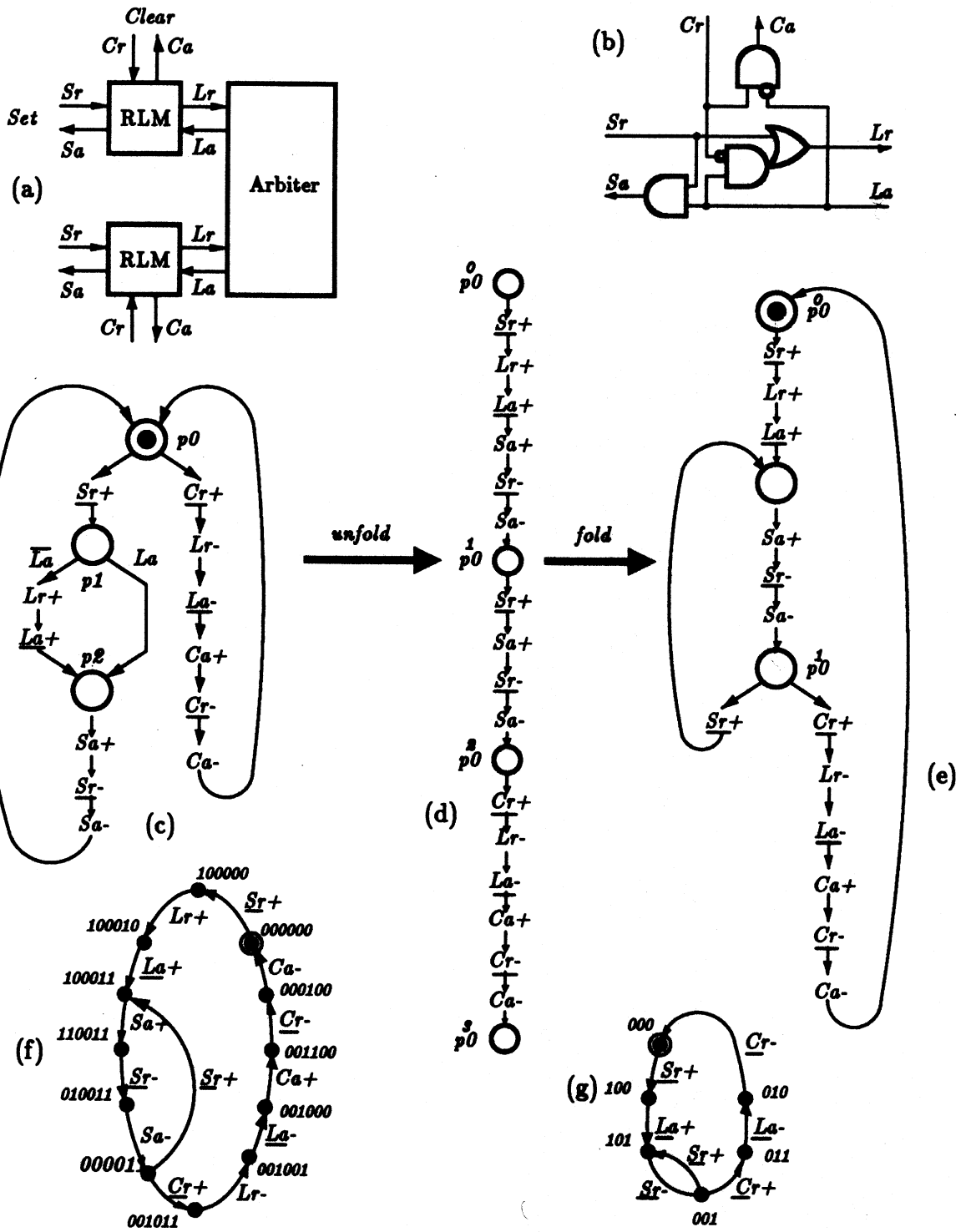


Figure 9.4: (a) The block diagram of the RLM and its (b) logic implementation. Its STG specification (c) is unfolded into a complete process (d), then folded back into the STG in (e). (f) The state graph of the STG in (e). (g) The control state graph.

traversing the left branch of the net, with $\langle L_a, 0\rangle$ being true. The segment between p_0^1 and p_0^2 corresponds to the left branch of the net, but with $\langle L_a, 1\rangle$ being true. The last segment between p_0^2 and p_0^3 corresponds to the right branch of the net.

This process contains instances of every element in the net and hence satisfies the completeness conditions described earlier. It is folded to produce the STG/IC shown in Fig. 9.4e as follows.

- Conditions p_0^0 and p_0^3 are merged together to form a strongly connected net.
- Conditions p_0^1 and p_0^2 are merged together because $[p_0^1, p_0^2]$ forms a complementary set. Then, to complete the folding, events input to p_0^1 and p_0^2 corresponding to transitions S_{a-}, S_{r-} and S_{a+} are also merged together.

Fig. 9.4f is the state graph of the STG/IC in Fig. 9.4e. This state graph can be decomposed to produce smaller state graphs for non-input signals $\{S_a, C_a, L_r\}$. From the STG/NC in Fig. 9.4c, we find

$$I(S_a)=\{L_a, S_r\}$$
$$I(C_a)=\{L_a, C_r\}$$
$$I(L_r)=\{L_a, S_r, C_r\}.$$

The logic implementation can be carried out by first deriving the contracted state graph for every non-input signal directly from the state graph given in Fig. 9.4f and then determining its logic equation. The final implementation is given in Fig. 9.4f; its derivation can be easily verified.

9.3 Properties of STG/NCs

In this section, we examine a number of relevant properties of STGs with non-input choices. One of the important issues to be addressed is the behavioral equivalence between a STG/NC and its expanded STG/IC; this will be managed by showing that the two STGs are equivalent iff they unfold into the same *complete* process. Another issue is the liveness property of STG/NCs; by examining this property we will be able to provide some insights into the construction of STG/NCs from initial informal specifications. One

last issue is how to characterize the temporal relation in such nets, i.e. how to determine whether two transitions are ordered, concurrent or in conflict using previous techniques.

It will be seen that the key concept used is that of complete processes. As described earlier, even though a complete process is acyclic and contains only a typical record of a net's execution, it still can reproduce the total behavior of a net in the form of a state graph. Such a state graph is constructed by merging markings with the same state assignment. Due to this fact, complete processes of STGs contain more information than conventional processes of FC nets.

In the rest of this section, we will consider a STG/NC $\Sigma_J = \langle P, T, F, M_0; \lambda \rangle$ which can be unfolded into a complete process $PN = \langle B, E, H; \phi \rangle$. This complete process can then be folded to produce a expanded net $\Sigma'_J = \langle P', T', F', M'_0 \rangle$. Let $N_\Sigma = \langle P, T, F \rangle$ and $N_{\Sigma'} = \langle P', T', F' \rangle$ denote the underlying net structures of Σ_J and Σ'_J, respectively. Then by definition, both N_Σ and $N_{\Sigma'}$ are free-choice nets.

Behavioral equivalence between STG/NC and STG/IC

Below, we sketch the reasoning for establishing the behavioral equivalence between a STG/NC and its expanded STG/IC: when Σ_J is successfully expanded into Σ'_J, their behaviors are equivalent. Two nets are ***behaviorally equivalent*** iff they have the same trace set or equivalently, the same finite automaton.

The key point is to show that if two nets are behaviorally equivalent, they must unfold into the same complete process. Suppose that

- Σ_J can be unfolded into a complete process PN and the latter folded back into the expanded net Σ'_J.
- Σ'_J can be unfolded into another complete process PN'.

Then it is easy to see that it is possible to choose a transition sequence of control signals such that the unfolding of $N_{\Sigma'}$ and N_Σ yield the same process, i.e. PN and PN' are identical; such a control sequence is recorded in the control state graph (CSG) described earlier. Hence, we can conclude that if Σ_J can be expanded into a live-safe net Σ'_J then they are behaviorally equivalent.

Liveness of STG/NCs

Recall that earlier, we defined liveness of STGs in terms of their state graphs: a STG is live iff its state graph is strongly connected. For STG/NCs, the same definition can be employed. Since the state graph of a STG/NC is obtained through its expanded net, a STG/NC is live iff its expanded net is live. The expansion algorithm successfully produces a live expanded net if the original STG/NC can be unfolded into a complete process. Thus, we have the following result.

Lemma 9.10 *A STG/NC is live iff it can be unfolded into a* complete *process.*

Hence in order to verify that a STG/NC Σ_J is live, one may need to apply the net-unfolding algorithm and check if the unfolded process is complete. If so, by folding the process, a live expanded net is then obtained. However, often times it is more convenient to use a certain necessary liveness condition directly on a STG/NC before unfolding. If this necessary condition is not satisfied by the STG/NC then it cannot be unfolded into a complete process. We study such a condition through an example below.

Consider a FC net as shown in Fig. 9.5a. This net is safe but *not* live because only two MG-components resulting from MG-reduction are live-safe, as indicated in Fig. 9.5c. Let's consider a firing sequence which leads to deadlock in Fig. 9.5a: in marking $\{p_1, p_2\}$, transitions t_1, t_2, t_3 and t_4 are all enabled, if t_1 and t_4 are chosen (nondeterministically) to fire, then p_3 and p_6 are marked. At this point the net's operation halts as there is no token to enable t_5 and t_6. Suppose now that instead of allowing the choices to be nondeterministic, we require that whenever places $\{p_1, p_2\}$ are marked, the pair $\{t_1, t_3\}$ be chosen to fire together, and similarly $\{t_2, t_4\}$ together. In this case, the net's operation is *live and safe.* This is one basic motivation behind the use of controlled choices. The STG/NC corresponding to this case is shown in Fig. 9.5b: output arcs of places $\{p_1, p_2\}$ are labeled with $\langle a, 0\rangle$ and $\langle a, 1\rangle$; in addition, transitions t_5 and t_6 are interpreted as $t_5 = a_-$ and $t_6 = a_+$.

The following lemma states a *necessary* condition on the underlying net of a STG/NC in order for it to be live-safe.

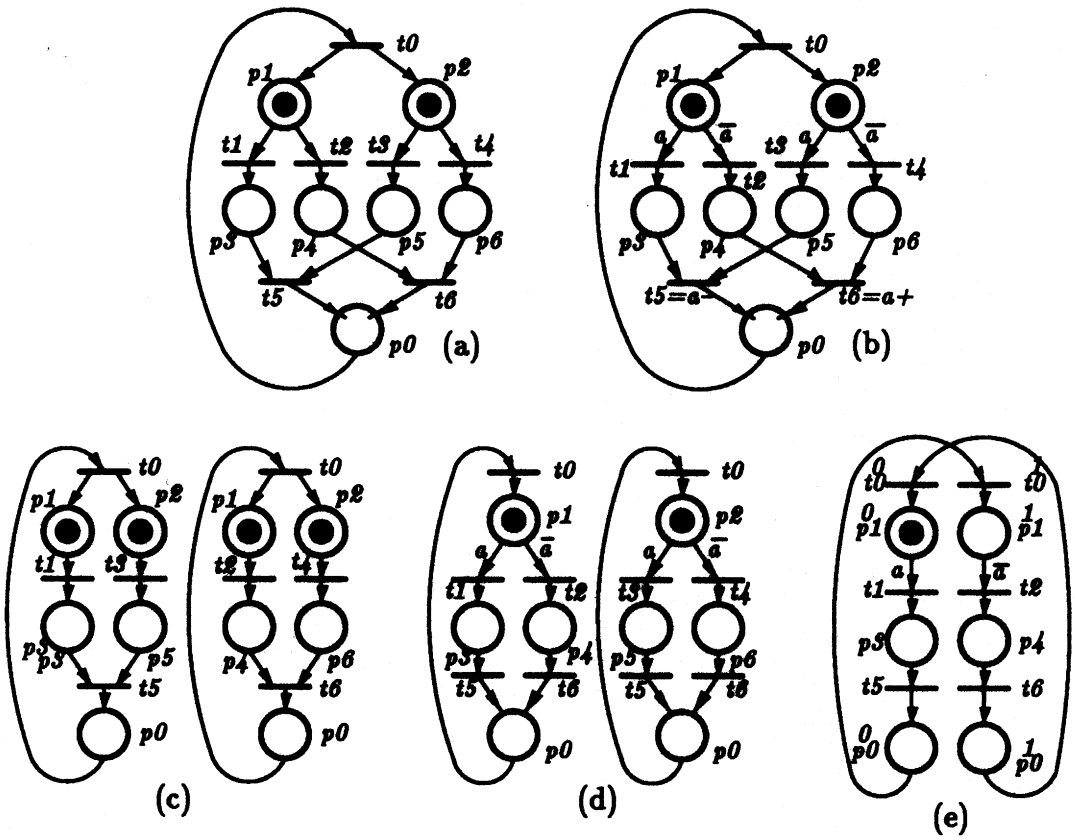


Figure 9.5: (a) A non-live FC net and (b) a STG/NC with the same underlying net. (c) Two live-safe MG-components. (d) Two live-safe SM-components, one of which is expanded into a simple cycle (e).

Lemma 9.11 *Let Σ_J be a STG/NC and N_Σ be its underlying* free-choice *net. Then STG/NC is live-safe only if there exists a set of MG-components which cover N_Σ.*

In a similar fashion, we see that only two SM-components resulting from SM-reduction on Fig. 9.5b are live-safe; they are depicted in Fig. 9.5d. Notice that each of these SM-components is itself a controlled-choice net. These SM-components can be expanded into simple cycles; Fig. 9.5e illustrates the case for the SM-component on the left of the figure. Let us consider a necessary condition for this controlled-choice net to be live. Obviously, it is live if its expansion yields a simple cycle as just mentioned. This condition in turns requires that a_+ not be ordered with t_1, and a_- not ordered with t_2. For suppose that $t_5 = a_+$ in Fig. 9.5d, then a_+ is ordered with t_1 and the simple cycle $p_0 t_0 p_1 t_1 p_3 t_5 p_0$ (where $t_5 = a_+$) will contain an arc labeled with $\langle a, 1 \rangle$. In which case, the control variable $\langle a, 1 \rangle$

will always be true, thus restricting the net's operation in this simple cycle and the other branch of the net can never be activated, resulting in non-liveness. Hence, for every controlled-choice place p and transition t such that $\lambda(p,t) = \langle a,1\rangle$ (or $\langle a,0\rangle$), the above condition requires that there exist no simple cycle containing t and a_+ (or a_-).

In a STG/NC, we require that for every place p and transition $t \in p\cdot$ such that p is a controlled-choice place and $\lambda(p,t) \in \{\langle a,0\rangle, \langle a,1\rangle\}$, p not be concurrent with any transition in $\{a_+, a_-\}$. Also, since t is not allowed to belong to the same simple cycle with $\{a_+, a_-\}$, it follows that t and a_+ must be ***in conflict*** in the underlying free-choice net (of the STG/NC).

Construction of STG/NCs

So far in this chapter, we have discussed the expansion algorithm for converting a STG/NC into an expanded net which corresponds to a LSFC net. That discussion assumes that a "correct" STG/NC is given. However, the construction of a STG/NC specification is generally not an easy task, because such a specification may involve complex interactions between choices and concurrency.

The above illustrative examples provide us with some insights for the construction of STG/NCs. As evidenced throughout this chapter, one of the key concepts is that of a complete process. Besides specifying all possible ***concurrent*** behavior of a system, it also contains extra information allowing one to reconstruct the ***free-choices***. Hence in the following procedure for STG/NC construction, we start with a process.

(*a*) ***Build a complete process:*** Construct an initial process according the concurrent behavior of a system. This process must contain segments representing possible choices of actions. The process can then be made ***complete***, possibly by adding extra internal transitions.

(*b*) ***Select control variables:*** The complete process can be folded into a free-choice net, and arc labels corresponding to control choices can be added to obtain a STG/NC.

(*c*) ***Check for live-safeness:*** The necessary conditions for live-safeness for such a STG/NC can be verified as follows. The STG/NC is live-safe only if (i) its underlying net is free-choice, (ii) there exists a set of live-safe MG-components which cover it, and

(iii) control variables are chosen such that every MG-component in this set can be activated at some marking reachable from the initial marking.

Temporal Relation in STG/NC

Previously, we have provided a syntactic characterization of the temporal relation for LSFC nets. However, due to the presence of controlled choices, this characterization does not apply directly to STG/NCs. As discussed earlier, the introduction of controlled choices into a STG imposes further restrictions on the set of firing sequences of the net. We will show that *such restrictions only affect the* sequential *but not the* non-sequential *behavior of the net, and concurrency is preserved.*

Let Σ_J be a STG/NC and Σ'_J be its expanded net; their underlying net structures are denoted by N_Σ and $N_{\Sigma'}$. Since all arc labels are not considered in N_Σ, both N_Σ and $N_{\Sigma'}$ are FC nets. Let $\mathbf{tr} = \mathbf{li} \cup \mathbf{co} \cup \mathbf{cf}$ and $\mathbf{tr}' = \mathbf{li}' \cup \mathbf{co}' \cup \mathbf{cf}'$ be the temporal relations defined in N_Σ and $N_{\Sigma'}$, respectively. Note that $\mathbf{tr}$ is the temporal relation in the *underlying* FC net of Σ_J; it is *not* the temporal relation in the STG/NC, which is really given by $\mathbf{tr}'$. In the following we study the relationship between $\mathbf{tr}$ and $\mathbf{tr}'$.

We can determine the temporal relation $\mathbf{tr}'$ of the STG/NC Σ_J from its expanded net Σ'_J which is a STG/IC. The underlying net $N_{\Sigma'}$ of the expanded net is a FC net and thus previous characterizations of the temporal relation apply.

First, note that a prerequisite for two transitions in the expanded net $N_{\Sigma'}$ to be concurrent is that they are concurrent in N_Σ. This is because the expanded net is obtained by unfolding N_Σ into a process and then folding the latter into $N_{\Sigma'}$. In a process, all conflicts must have been resolved, and concurrency between instances of transitions must be preserved. Hence concurrency is preserved: $\mathbf{co}' \subseteq \mathbf{co}$.

The situation with ordering and conflict is slightly more complicated. The example given in Fig. 9.5 demonstrates that due to controlled-choices, the number of transitions in conflict in a STG/NC (Fig. 9.5b) cannot be more than that in its underlying FC net. In the expanded net, a controlled-choice place in a STG/NC is eliminated during unfolding. Hence: $\mathbf{dc}' \subseteq \mathbf{dc}$.

On the other hand, Fig. 9.5e shows that a SM-component of a STG/NC can be

expanded into a simple cycle. This implies that:

(i) Two transitions are in conflict in the expanded net $N_{\Sigma'}$ only if they are also in conflict in N_Σ: $\mathbf{cf}' \subseteq \mathbf{cf}$.

(ii) On the other hand, if two transitions are ordered in the expanded net $N_{\Sigma'}$ then in N_Σ, they may be either ordered or in conflict: $\mathbf{li}' \subseteq \mathbf{li} \cup \mathbf{cf}$.

(iii) If two transitions are ordered in N_Σ then in the expanded net $N_{\Sigma'}$ they must also be ordered: $\mathbf{li} \subseteq \mathbf{li}'$.

(iv) If two transitions are in conflict in N_Σ then in the expanded net $N_{\Sigma'}$ they may be either ordered or in conflict: $\mathbf{cf} \subseteq \mathbf{cf}' \cup \mathbf{li}'$.

From these facts, we can deduce an equality (vii) as follows:

$$\begin{array}{lll}
\text{(iv)} & \Rightarrow & \mathbf{cf} \subseteq \mathbf{li}' \cup \mathbf{cf}' \\
 & \Rightarrow & \mathbf{li} \cup \mathbf{cf} \subseteq \mathbf{li} \cup \mathbf{li}' \cup \mathbf{cf}' \\
\text{(iii)} & \Rightarrow & \mathbf{li} \cup \mathbf{cf} \subseteq \mathbf{li}' \cup \mathbf{cf}' \qquad \text{(v)} \\
\text{(i) \& (ii)} & \Rightarrow & \mathbf{li}' \cup \mathbf{cf}' \subseteq \mathbf{li} \cup \mathbf{cf} \cup \mathbf{cf} = \mathbf{li} \cup \mathbf{cf} \qquad \text{(vi)} \\
\text{(v) \& (vi)} & \Rightarrow & \mathbf{li}' \cup \mathbf{cf}' = \mathbf{li} \cup \mathbf{cf} \qquad \text{(vii)}
\end{array}$$

The above results are summarized in the following lemma, which simply states that when a FC net is converted into a STG/NC by adding arc labels, both sequentiality and non-sequentiality are preserved.

Lemma 9.12 *Let Σ_J and Σ'_J be a STG/NC and its expanded net; N_Σ and $N_{\Sigma'}$ their respective underlying nets, as defined above. Then*

$$\begin{array}{l}
\mathbf{co}' \subseteq \mathbf{co} \\
\mathbf{li}' \cup \mathbf{cf}' = \mathbf{li} \cup \mathbf{cf}.
\end{array}$$

The above result ensures that (i) if two transitions are not concurrent in the underlying net N_Σ of a STG/NC, then they will not be concurrent at all, and (ii) if they are ordered or in conflict in N_Σ then they will also be either ordered or in conflict in the expanded net $N_{\Sigma'}$.

9.4 Summary

In this chapter, we have presented an algorithm for converting STG specifications with non-input choices to ones which contain only input choices. The most crucial underlying

concept of STG/NCs and this expansion algorithm is concerned with the behavioral (i.e. firing sequence) semantics of nets: STGs are considered as high-level representations of state graphs. In particular, the use of controlled-choices can be thought of as a further restriction of the trace set derived from a STG. Similarly, the expansion algorithm basically manipulates the state graph from a higher level and it performs the splitting of states with non-input choices. By requiring that arc labels of a controlled-choice place never hold simultaneously at the moment that place is marked, we can guarantee that split states have distinct binary representations.

One important concept in this chapter is that of a complete process. Even though it is acyclic, such a process contains enough information concerning the complete behavior of the system, so that from it choices between alternate control events can also be deduced.

One fundamental requirement in order for a STG/NC to unfold into a process is that transitions of a control variable not be concurrent with transitions which are output transitions of the controlled-choice places. This is a fundamental requirement, because the only way to determine the state of a control variable, say j, at a marking (at which a controlled-place, say p, is marked) is for the transitions of the variable j to be ordered with p; otherwise, it is impossible to tell exactly.

Chapter 10

Suggestions for further research

In this short chapter, we suggest areas for further research. Some of the extensions to the STG model to be discussed are:

- Development of high-level hardware description languages for the specification of self-timed systems.
- Techniques for composition and decomposition of self-timed control circuits based on STGs.
- Performance evaluation and optimization of control circuits synthesized from STGs.
- Pragmatic issues concerning the use of STGs in silicon compilation.

High Level Hardware Description Language

Even though the design approach based on STGs allows the direct and efficient synthesis of control circuits, one of its shortcomings is that STGs are rather low level: A behavior specification of a control circuits in terms of STG requires the tedious enumeration of its exact behavior at the level of signal transitions—this is especially true for control circuits whose operation involves both concurrency and choices, such as the FIFO module described in Chapter 8. To alleviate this problem, we need (i) more abstract specifications which can be refined iteratively to produce more detailed descriptions and (ii) a design methologoly for self-timed systems. A reasonable strategy involves the use of a high level description language to serve both purposes.

- A high level language provides an abstraction of detailed implementation which can be refined subsequently. For instance, at the more abstract level of specification, it should be possible to represent parts of a STG as single events; signal transitions specifying the communication protocol between control modules should also be represented as single synchronization events.
- A high level language can be used to enforce a design style on the use of STGs. In particular, the syntax of the language can be used to disallow "bad" constructions. For instance, in the CSP-like language proposed by Martin [30], each module is represented by a sequential process; parallelism is achieved by having many sequential processes commnunicating with each other through synchronization. Obviously, the design style imposed by such language is that each module can perform only sequential operations; concurrency results from the interconnection of sequential modules operating in parallel.

STGs allow the specification of control modules with inherent parallel operation (such as the FIFO module in Chapter 8) as well as those with sequential operation and choices (the RLM in Chapter 9). Thus the description language for STGs should not prohibit the specification of concurrent operation internal to modules. Our strategy is to identify a design approach for STGs and then develop language constructs which allow the description of hardware control circuits according to the design approach. Some of the possible candidates for our hardware description language include *path expressions* [1] and *PADL* [29], to name a few. The semantics of path expressions in terms of Petri nets has been studied in [6]. On the other hand, PADL is designed expressly for the description of packet architectures which are related to *data flow graphs*. A net semantics can be easily developed for a subset of PADL.

Composition and Decomposition

The design of a system is unmanageable unless there is a hierarchical approach which allows the system to be decomposed into components, each desgined separately and then composed together to form the whole system. The important issue here is the correctness of such systems when the decomposition and composition steps are taken: how to guarantee that the behavior of the original system before decomposition is the same as that of one

composed from interconnecting modules, each corresponding to a decomposed part of the system.

The firing sequence (trace) semantics can help: properties of composed and decomposed systems can be reasoned using traces and operations on traces. Many ideas and mathematical results developed for CSP can be applied with some modification; these include high-level specifications, proof rules, etc.

This thesis has developed a useful result concerning the equivalence between the behavior of a FC net and the aggregate behavior of a set of contracted nets. This is an instance of decomposition and composition of nets. However, it differs from composition/decomposition in general in that the former starts out with a net whose live-safeness properties are known ahead of time, while in the latter, these properties have to be proven for the system composed from modules.

Performance Evaluation and Optimization

STGs allow the specification and direct implementation of speed-independent control circuits; circuits obtained from STGs are hazard free regardless of changes in the delays of logic components. Nevertheless, it is always important to be able to determine the timing performance of control circuits so that their critical parts can be optimization when needed.

One can take the simple view of a STG as specifying the timing constraints between signal transitions; a constraint such as $i_* \rightarrow j_*$ is implemented as a logic element with input i and output j. Thus a time delay d can be associated to the arc $i_* \rightarrow j_*$ to represent the delay between transitions of signals i and j. This delay can be determined from the logic implementation.

By calculating the delay values and associate them to all causal arcs in a STG, a timed model is obtained. Techniques in *Timed Petri nets* [33] can then be applied directly for the purpose of performance analysis and optimization.

One specific type of optimization is the transformation of a speed-independent implementation into one which is only hazard-free. Recall that speed-independence means that a circuit is hazard-free for *any* combination of delays of logic gates. By determining the

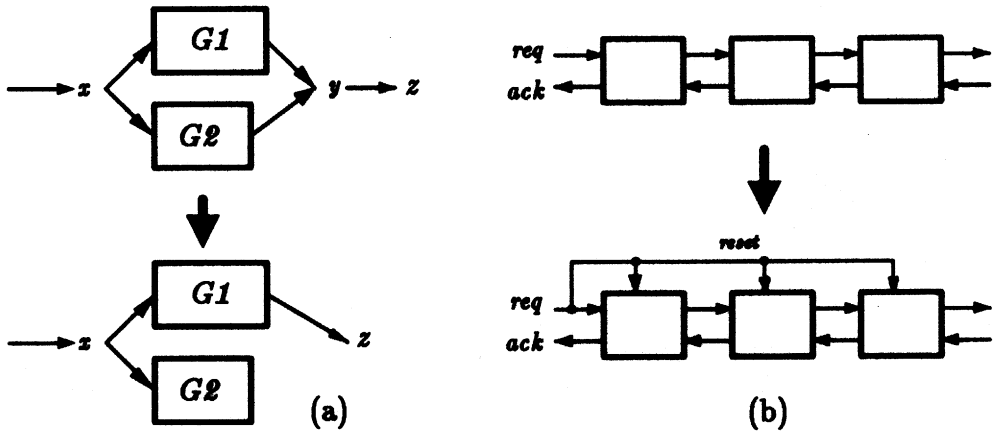


Figure 10.1: Transforming a speed-independent circuit into a hazard-free circuit. (a) By removing synchonization constraints. (b) By resetting modules in parallel.

bounds on their delays, one can remove some of the causal constraints for synchronation from the STGs without affecting the correct operation of the circuit. For example, in the STG of Fig. 10.1a, the delays of the subgraph G_1 and G_2 can be determined from their logic implementation to be d_1 and d_2, respectively. If $\min d_1 > \max d_2$ then the transition y which synchronizes the actions of G_1 and G_2 can be removed. Such an optimization results in a saving of logic gates and an improvement in speed.

Another type of optimization is a well-known technique for reducing the total delay of an operation cycle using the reset signaling protocol, as illustrated in Fig. 10.1b. If a number of non-pipelined modules are connected in tandem, the input *request* to the leftmost module must propagate through the all of them before an *acknowledge* is transmitted back. In the reset phase of the signaling protocol, the input *request* is reset and it is allowed to propagate through the modules. The operation is completed when the *acknowledge* is reset by the leftmost module. The operation during the reset phase of the cycle does not involve any useful work but only resetting the modules to their previous quiescent state. The total delay can be reduced by connected the leftmost input *request* signal to *all* modules in the chain, so that they can be reset to their initial state in parallel. The time required for reseting can thus be reduced to a constant. This technique works well in practice even though the circuit can no longer be guaranteed to be speed-independent. In CMOS technology, there is a natural way to implement this technique using *domino* logic, as demonstrated in [10].

Pragmatics

One of our goals is to use STGs as a formal specification of control circuits in a hardware architecture for self-timed systems. It is perfectly feasible to build design-aid tools for the automatic design of self-timed systems. Some practical issues of concern are the specification and design of data paths, and the choice of hardware medium for VLSI implementation.

The data-path components of such an hardware architecture can also be specified in terms of Petri nets; they can be implemented using dual-rail coding or matched delay techniques. A hardware implementation based on regular arrays similar to Patil's asynchronous logic arrays [38] seems to be advantageous and practical.

Bibliography

[1] T. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling Path Expressions into VLSI circuits. In *Proceedings of the 12th Symposium on Principles of Programming Languages*, ACM, January 1985.

[2] G. Berthelot and G. Roucairol. Reduction of Petri Nets. In *Lecture Notes in Computer Science 45*, Springer-Verlag, 1976.

[3] E. Best. Concurrent Behaviour: Sequences, Processes and Axioms. In *Lecture Notes in Computer Science 197*, Springer-Verlag, 1984.

[4] E. Best and K. Voss. Free Choice Systems have Home States. *Acta Informatica*, (21), 1984.

[5] D. L. Black. On the existence of delay-insensitive fair arbiters: trace theory and its limitations. *Journal of Distributed Computing*, 1:205–225, 1986.

[6] H. R. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science 26*, pages 89–102, Springer-Verlag, 1974.

[7] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions On Computers*, C-22, April 1973.

[8] T. Chu and L. A. Glasser. Synthesis of self-timed control circuits from graphs: an example. In *Proceedings of the International Conference on Computer Design*, IEEE, New York, Oct 1986.

[9] T. Chu and C. K. Leung. Design of VLSI asynchronous FIFO queues for packet communication networks. In *Proceedings of the International Conference on Parallel Processing*, IEEE, August 1986.

[10] T. Chu, C. K. Leung, and T. S. Wanuga. A design methodology for concurrent VLSI systems. In *Proceedings of the International Conference on Computer Design*, IEEE, New York, Oct 1985.

[11] W. A. Clark. Macromodular computer systems. In *Proceedings of the Spring Joint Computer Conferences*, AFIP, 1967.

[12] F. Commoner *et al.* Marked directed graphs. *Journal of Computer and System Sciences*, 5, 1971.

[13] G. Couranz and D. Wann. Theoretical and experimental behavior of synchronizers in the metastable region. *IEEE Transactions On Computers*, C-24:604–616, 1975.

[14] J. B. Dennis. Data flow supercomputers. *IEEE Computers*, November 1980.

[15] J. B. Dennis. Modular, asynchronous control structures for a high performance processor. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 55–80, ACM, 1970.

[16] J. B. Dennis and S. S. Patil. The description and realization of digital systems. In *Digest of Papers of the Sixth Annual IEEE Computer Society International Conference*, pages 223–226, 1972.

[17] J. B. Dennis and S. S. Patil. Speed-independent asynchronous circuits. In *Proceedings of the Fourth Hawaii International Conference on System Sciences*, pages 55–58, 1971.

[18] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[19] L. D. Dill and E. Clarke. Automatic verification of asynchronous circuits using temporal logic. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, May 1985.

[20] A. D. Friedman and P. Menon. *Theory and Design of Switching Circuits*, chapter 4. Computer Science Press, 1981.

[21] L. A. Glasser. Synchronizer failure in A/D converters. MIT VLSI Memo. 85-276, October 1985.

[22] L. A. Glasser and D. W. Dopperpuhl. *The Design and Analysis of VLSI Circuits*, chapter 6. Addison Wesley, 1985.

[23] U. Goltz and W. Reisig. Processes in Place/Transition nets. In *Proceedings of ICALP'83*, Springer-Verlag, 1983.

[24] M. Hack. *Analysis of Production Schemata by Petri Nets.* TR 94, Project MAC, Massachusetts Institute Technology, Cambridge, Mass, 1972.

[25] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[26] L. A. Hollarr. Direct implementation of asynchronous control circuits. *IEEE Transactions On Computers*, C-3(12), December 1982.

[27] A. W. Holt *et al. Final report for the project-development of theoretical foundations for description and analysis of discrete information systems.* Technical Report, Mass. Comp. Assoc., Wakefield, Mass, 1974.

[28] R. M. Keller. Toward a Theory of Universal Speed-Independent Modules. *IEEE Transactions On Computers*, C-23(1):21–32, January 1974.

[29] C. Leung and W. Lim. *A Packet Architecture Description Language.* MIT/LCS/TR 306, Laboratory for Computer Science, MIT, Cambridge, Mass, 1982.

[30] A. J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Journal of Distributed Computing*, 1:226–234, 1986.

[31] A. Mazurkiewics. Semantics of Concurrent Systems: A Modular Fixed-Point Trace Approach. In *Lecture Notes in Computer Science 188*, Springer-Verlag, 1984.

[32] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 7, System Timing. Addison Wesley, 1981.

[33] M. K. Molloy. Perfomance analysis using Stochastic Petri Nets. *IEEE Transaction on Software Engineering*, SE-11(4):913–917, September 1982.

[34] C. E. Molnar, T. Fan, and F. U. Rosenberger. Synthesis of Delay-Insensitive Modules. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, May 1985.

[35] D. E. Muller. Asynchronous Logics and Application to Information Processing. In *Switching Theory in Space Technology*, Stanford Univ. Press, California, 1959.

[36] D. E. Muller. The general synthesis problem for asynchronus digital networks. In *Conference record of the Eight Annual Symposium on Switching and Automata Theory*, 1967.

[37] D. E. Muller and W. S. Bartky. *A theory of asynchronous circuits.* Volume 29 of *the Annals of the Computation Laboratory of Harvard University*, Harvard Univ. Press, Cambridge, Mass, 1959.

[38] S. S. Patil. *An asynchronous logic array.* Tech. Memo. 62, Project MAC, Massachusetts Institute Technology, May 1975.

[39] J. L. Peterson. *Petri net theory and the modeling of systems.* Prentice Hall, 1981.

[40] W. W. Plummer. *Asynchronous Arbiters.* CSG Memo 56, Project MAC, Massachusetts Institute Technology, February 1978.

[41] W. Reisig. *Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Heidelberg, 1985.

[42] M. Rem. Concurrent Computations and VLSI Circuits. In D. Bjørner, editor, *Formal Description of Programming Concepts - II*, IFIP, North-Holland Pub. Co., 1983.

[43] G. Rozenberg. Behaviour of elementary net systems. In *Advanced Course on Petri Nets*, September 1986. To appear in *Lecture Notes in Computer Science* 1987.

[44] G. Rozenberg and P. S. Thiagarajan. Petri nets: basic notions, structure, behaviour. In *Lecture Notes in Computer Science 224*, Springer-Verlag, 1986.

[45] H. Schols. *A formalisation of the foam rubber wrapper principle.* Master's thesis, Eindhoven University of Technology, 1985.

[46] C. L. Seitz. Asynchronous machines exhibiting concurrency. In *Record of the Project MAC conference on concurrent systems and parallel computations*, ACM, 1970.

[47] M. J. Stucki and J. R. Cox. *Synchronization Strategy.* TR WUCS-79-1, Dept. of Computer Science, Washington Univ., Saint Louis, Mo, 1979.

[48] E. T. Tanner and C. Mead. A correlating optical motion detector. In P. Penfield, editor, *Proceedings of the 1984 Conference on Advanced Research in VLSI, M.I.T.*, pages 57–64, Artech House, MA, 1984.

[49] P. S. Thiagarajan and K. Voss. A fresh look at free-choice nets. *Information and Control*, 61(2), 1984.

[50] K. Tognoni. Metastable problems in A/D converters. S.B. thesis, Department of EECS, MIT, 1985.

[51] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Dept. of Mathematics and Computing Science, Eindhoven Univ. of Technology, 1984.

[52] S. H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions On Computers*, C-20(12), December 1971.

[53] J. van de Snepscheut. *Trace Theory and VLSI Design. Lecture Notes in Computer Science 200*, Springer-Verlag, 1985.

[54] R. E. Zippel. *Contemporary MOS Circuits.* Lecture notes for Course 6.721, Department of EECS, MIT, 1984.

CS-TR Scanning Project
Document Control Form

Date : 4/26/96

Report # LCS-TR-393

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 198 (204-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-198) UNNUMBERED TITLE PAGE, i-viii, 1-189	
(199-204) SCANCONTROL, COVER, SPINE, TRGT'S(3)	

Scanning Agent Signoff:
Date Received: 4/26/96 Date Scanned: 4/26/96 Date Returned: 5/2/96

Scanning Agent Signature: Michael W. Cooper

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 04/26/1996

M.I.T. Libraries
Document Services